



Université
de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

M. MUHAMMAD ADNAN

le jeudi 21 novembre 2013

Titre :

ANALYSE PIRE CAS EXACT DU RESEAU AFDX

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (I.R.I.T.)

Directeur(s) de Thèse :

M. CHRISTIAN FRABOUL

M. JEAN LUC SCHARBARG

Rapporteurs :

M. JEAN-JACQUES LESAGE, ECOLE NORMALE SUPERIEURE DE CACHAN

M. THIERRY DIVOUX, UNIVERSITE NANCY 1

Membre(s) du jury :

M. GUY JUANOLE, UNIVERSITE TOULOUSE 3, Président

M. CHRISTIAN FRABOUL, INP TOULOUSE, Membre

M. JEAN-LUC SCHARBARG, INP TOULOUSE, Membre

M. JEROME ERMONT, INP TOULOUSE, Membre

M. JUAN LOPEZ, AIRBUS FRANCE, Membre

Mme ISABELLE AUGÉ BLUM, INSA LYON, Membre

Acknowledgments

With immense pleasure, I acknowledge the help, guidance and support received from various quarters during the course of my PhD studies. Pursuit of PhD degree is a long journey which is very hard to accomplish without the help of others. One of the joys of completion is to look over the journey past and remember all the friends and family who have helped and supported me along this long but fulfilling road. Hence, I hereby express my appreciation to one and all.

My sincere and unbounded gratitude to my advisor *Christian Fraboul* and co-advisors, *Jean-Luc Scharbarg* and *Jérôme Ermont* for their support and guidance. Their vision and suggestions throughout the thesis period, from choosing the problem, solving it and its presentation, has been fundamental in completion of the thesis. The joy and enthusiasm they have for the research, was contagious and motivational for me, even during tough times. Sir, you have been a source of motivation throughout. I am also very grateful to the remaining members of my dissertation committee: Professor Guy JUANOLE president of the jury, Jean-Jacques LESAGE and M. Thierry DIVOUX, my honorable reporters, Mme AUGÉ-BLUM Isabelle and Mr Juan LOPEZ from airbus Toulouse, our industrial partner. Their academic support and input and personal cheering are greatly appreciated. Thank you all.

The members of the research laboratory at *ENSEEIH/IRIT* have contributed positively to my personal and professional learning. The team as a whole, has been a source of friendships as well as good advice and collaboration. My colleagues here at *ENSEEIH/IRIT Lab* deserve more than a mention. I acknowledge their support and help during my stay in the lab. Their companionship, help and guidance in both academic and logistic issues is worth the praise. The lab wouldn't be as much fun to stay in without you all. I am grateful to *Christelle Jacob* and her parents for their hospitality and support on numerous occasions during my stay in France. I felt at home with you all and will cherish these days all my life.

Every one I met at different academic conferences and gatherings, deserves an honorable mention for their discussions and inputs on everything academic or otherwise. I have definitely learned and improved my work with all your suggestions.

My special thanks to all the administrative and secretarial staff, both at the lab and university and at my embassy and people back home, for doing all the necessary work they did for me, so that I was able to concentrate on my research without distractions.

This section would not be complete without the mention of *beloved parents* whose love and

affection will be treasured throughout. My family members have always been a source of constant support, encouragement and inspiration. Thanking them would be an understatement. So I would like to conclude my acknowledgments by dedicating this work to my parents.

Muhammad Adnan

University of Toulouse

November 2013

Executive Summary (Resume)

Since the last few decades, Information and Communication Technology (ICT) systems are evolving rapidly and they are becoming more and more popular and omnipresent. This trend also exist in aviation domain. Now a days, all modern aircraft have complex suit of on-board electronic devices used for various purposes commonly referred to as Avionics Systems. Avionics systems of an aircraft are used in a wide variety of different applications such as flight control, instrumentation, navigation, communication etc. These avionics systems need to communicate between themselves and exchange data, hence building "Avionics networks". Over the years, demand for data exchange has risen rapidly and avionics networks have evolved from dedicated links to shared buses to switched networks such as Avionics Full-Duplex Switched Ethernet (AFDX). AFDX is a data network for safety critical applications that utilizes dedicated bandwidth while providing deterministic Quality of Service (QoS). AFDX is based on IEEE 802.3 Ethernet technology and utilizes commercial off-the-shelf (COTS) components. It is described specifically by Part 7 of the ARINC 664 Specification, as a special case of a profiled version of an IEEE 802.3 network per parts 1 & 2, which defines how Commercial Off-the-Shelf networking components will be used for future generation Aircraft Data Networks (ADN). The six primary aspects of AFDX include full duplex, redundancy, deterministic, high speed performance, switched and profiled network. Like any other communication network being used on-board an aircraft, it is very important to know the temporal aspects of data flow on AFDX network, such as communication delay from the source to the destination. These end to end communication delays are important to determine because they are used to certify avionics systems of the aircraft. In this context, the main objective of this thesis is to provide methodologies of finding exact worst case communication delays of AFDX network.

To achieve this goal, different tools and approaches have been analyzed and compared with existing techniques. New approaches and algorithm were also developed during the research work of this thesis. At present two main techniques are being used for end to end delay analysis of AFDX network. These are Network Calculus and Trajectory approach. Both of these are pessimistic in their results and give us a sure upper bound on the end to end communication delays instead of exact values. Network Calculus uses Min Plus algebra for its calculations. The pessimism in results have been reduced by using different techniques such as "grouping". Trajectory approach uses concept of "busy period" to calculate its bounds for end to end communication delays. In some cases Network Calculus has better results than the Trajectory approach while in other cases Trajectory approach gives better results. On average, results of Trajectory approach are tighter than Network Calculus approach and the margin varies depending upon

the VL path.

In order to evaluate exact end to end communication delays, Model checking has been used in the context of AFDX network. Before this research work, it was applied to AFDX network as a proof of concept on a simple configuration. During this work, we have explored Model checking for end to end communication delays of AFDX network in depth, with models reflecting the real configuration parameters, such as asynchronous behavior, packet sizes and BAG values. In this context, existing well established real time model checking tools were explored, such as UPPAAL and NuSMV. UPPAAL suits better for the end to end communication delays in AFDX network as compared to NuSMV because NuSMV can only handle pure discrete models. On the other hand UPPAAL does not have a symbolic representation for the discrete part of the state space and hence it limits the size of models that can be evaluated in reasonable time and computation resources. Still, we were able to evaluate AFDX network of considerably larger sizes than existing approach. We are able to find end to end communication delays of AFDX network with upto 32 VLs.

In order to overcome limitations of Model Checking approach, the work was done in the direction of exhaustive simulation using in house developed algorithms and tools based on these algorithms. The main reason for using this approach was to develop a tool from scratch which is specifically suited for the task of finding exact end to end communication delay of AFDX networks. In order to reduce state space for this exhaustive simulation approach, properties of the AFDX network were exploited and different algorithms were developed which ensure that we only consider cases which can be candidate for worst case end to end communication delays. The end result is encouraging and we were able to analyze large AFDX network configurations. We were also able to analyze part of a real life industrial configuration of the AFDX network with approximately 1000 VLs and 6400 paths. For more than 60% of these paths we were able to find exact end to end communication delays while for the rest we were able to find end to end communication delays which are close to worst case communication delays.

The results obtained from the tool developed during this research were compared with existing approaches. With exact end to end communication delays calculated by this tool, we can find exact pessimism in Network Calculus and Trajectory approaches. On average, Network Calculus is 13% pessimistic in its calculations while Trajectory approach is about 6% pessimistic in its calculations.

Keywords: AFDX Network, Model Checking, Worst Case Communication Delay, Exhaustive Simulation

List of Personal Publications

- [Adnan 2010b] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. *Model for worst-case delay analysis of an AFDX network using timed automata*. In Proc. of the 15th ETFA , Bilbao, pp.1-4, 13-16 Septembre 2010. doi: 10.1109/ETFA.2010.5641124 keywords: automata theory;avionics;delays;local area networks;scheduling;AFDX network;ARINC 664 standard;avionics full duplex switched Ethernet;end-to-end communication delays;local scheduling;timed automata;upper bounds;worst case delay analysis model. URL : <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5641124&isnumber=5640954> (Cited on pages 3, 5, 50 and 111.)
- [Adnan 2010c] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont, Christian Fraboul. Worst-case end-to-end delay analysis of switched Ethernet using timed automata. In : Junior Researcher Workshop on Real-Time Computing, Toulouse, IRIT, pp. 23-26,4-5 November 2010. keywords: automata theory;avionics;delays;local area networks;scheduling;AFDX network;ARINC 664 standard;avionics full duplex switched Ethernet;end-to-end communication delays;local scheduling;timed automata;upper bounds;worst case delay analysis model. (Cited on page 5.)
- [Adnan 2011a] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. *An improved timed automata model for computing exact worst-case delays of AFDX periodic flows*. In Proc. of the 16th ETFA , Toulouse, pp.1-4, 5-9 Septembre 2011. doi: 10.1109/ETFA.2011.6059162 keywords: automata theory;delays;local area networks;real-time systems;AFDX periodic flow;avionics switched Ethernet network;network calculus;timed automata model;trajectory approach;worst-case end-to-end communication delay;Aerospace electronics;Analytical models;Automata;Clocks;Computational modeling;Delay;Upper bound;AFDX network;Timed Automata;UPPAAL Modelling;Worst case delay analysis. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6059162&isnumber=6058966> (Cited on pages 5, 76, 77 and 79.)
- [Adnan 2011b] Muhammad Adnan, Jean-Luc Scharbarg and Christian Fraboul. *Minimizing the search space for computing exact worst-case delays of AFDX periodic flows*. In Proc. of the 6th SIES, Vasteras, pp.294-301, 15-17 June 2011. doi: 10.1109/SIES.2011.5953673 keywords: avionics;delays;local area networks;multi-access systems;search problems;switching networks;AFDX periodic flow;ARINC 664 standard;avionics full duplex switched Ethernet;search space minimization;virtual links;worst

case end-to-end communication delay;Aerospace electronics;Calculus;Computational modeling;Context;Delay;Silicon;Upper bound;AFDX network;Guided simulation;Schedulability analysis;Worst case delay analysis. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5953673&isnumber=5953643> (Cited on pages 5 and 58.)

- [Adnan 2012] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. An improved timed automata approach for computing exact worst-case delays of AFDX sporadic flows. In Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on, pp. 1-8, 17-21 Septembre 2012. doi: 10.1109/ETFA.2012.6489576 keywords: avionics;local area networks;telecommunication computing;AFDX sporadic flows;ARINC 664 standardised;avionics full duplex switched Ethernet;end-to-end communication delays;network calculus;periodic AFDX configurations;timed automata approach;worst-case delay computing;AFDX network;Timed Automata;UPPAAL Modelling;Worst case delay analysis. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6489576&isnumber=6489522> (Cited on pages 5 and 79.)

Contents

Acknowledgments	i
Executive Summary	iii
List of Personal Publications	v
1 Introduction	1
1.1 The Context	2
1.2 Contribution	4
2 Background: System Verification and AFDX Network	7
2.1 System Verification	7
2.1.1 Software Verification	10
2.1.2 Hardware Verification	10
2.1.3 Behavioral Verification	11
2.2 Model Checking	12
2.2.1 Formal Methods	12
2.2.2 Model-based Verification	13
2.2.3 History of Model Checking	15
2.2.4 Application of Model Checking in Networks	15
2.2.5 Characteristics of Model Checking	15

2.3	AFDX Network	18
2.3.1	History of Aircraft Data Networks (ADN)	19
2.3.2	Overview of AFDX	19
2.3.3	Virtual Links (VL)	21
3	State of the Art: Methods to Compute the Worst Case End to End Delays in an AFDX Network	25
3.1	Bounds of Worst Case End-to-End Communication Delays	26
3.1.1	Network Calculus	26
3.1.2	Trajectory Approach	32
3.1.3	Pessimism of Network calculus and Trajectory approach	38
3.1.4	Conclusion	40
3.2	Exact Worst Case End-to-End Communication Delays	40
3.2.1	Model Checking	40
3.2.2	Exhaustive Simulation	58
3.2.3	Conclusion	58
3.3	Conclusion	59
4	An Improved Method to Compute the Exact Worst Case End-to-End Delay using Timed Automata	61
4.1	Characteristics of a worst-case scenario	62
4.1.1	Definition of a scenario	62
4.1.2	Critical Instance Property	62

4.2	The modelling based on timed automata	65
4.2.1	Modelling the VLs	66
4.2.2	Modelling the Switches	72
4.2.3	Modelling the Synchronization	74
4.2.4	Utility Automata: modelling of the buffers	76
4.2.5	Utility Automata: end to end delay computation	77
4.3	Limits of the approach	79
4.4	Conclusion	81
5	A New Approach Based on Exhaustive Simulation to Compute the Exact Worst-Case End to End delays	83
5.1	Modelling of the network and a scenario	84
5.1.1	Nomenclature and definitions	85
5.1.2	Modelling of a scenario	86
5.1.3	Reducing the number of scenarios	87
5.2	Computing worst case end to end delays using sequences	88
5.2.1	Computation of delay and merging of sequences at a switch output port	89
5.3	Worst-case end to end delay computations on a simple AFDX network using sequences	90
5.3.1	Presentation of the system	90
5.3.2	Computing the worst case end to end delay of VL under study	90
5.3.3	Computation of the sequences generated at the input of switch $S2$	92
5.3.4	Computation of the resulting sequences at the output of switch $S2$	93

5.3.5	Computation of the sequences at the input ports of switch $S1$	94
5.3.6	Computation of the sequences at the output of switch $S1$	95
5.3.7	Computation of the sequences at the input ports of switch $S3$	96
5.3.8	Computation of the sequences at the output of switch $S3$	97
5.4	Evaluation of the sequence based approach	97
5.5	More Improvements and reduction in scenarios	99
5.5.1	Modeling of Sporadic traffic	101
5.5.2	Further reduction of scenarios	102
5.5.3	Candidate scenario for worst case delays	107
5.5.4	Algorithm to further reduce number of cases	107
5.6	Conclusion	111
6	Case Study	113
6.1	AFDX network system of industrial scale complexity	113
6.1.1	Understanding the complexity of industrial scale AFDX network	113
6.2	Software Architecture	120
6.3	Results of the Case Study	120
6.3.1	Comparison of results with Network Calculus and Trajectory approach . .	124
6.4	Conclusion	124
7	Conclusions and Prospective	127
7.1	Conclusions	127

7.2 Prospective	129
A Model Checking Overview	133
A.1 Classification	133
A.2 List of Model Checkers, Modeling Languages and Specification Languages	135
A.2.1 List of Modeling Languages	135
A.2.2 List of Property Specification Languages	138
A.3 Relevance/Application to AFDX Network	138
B Software Architecture	141
B.1 Software Architecture	141
B.1.1 Parser	142
B.1.2 Network Pruning	143
B.1.3 Load Balancer	144
B.1.4 Compute Module	145
B.1.5 Control Logic	146
Bibliography	147
Index	153
List of Abbreviations	155

List of Figures

1.1	The Ariane-5 crash	2
1.2	AFDX network delays analysis.	4
2.1	Schematic view of system verification process.	8
2.2	Schematic view of Model Checking process.	14
2.3	ARINC 429 vs AFDX architecture	20
2.4	AFDX network	22
2.5	AFDX virtual links.	23
2.6	AFDX switch architecture.	23
3.1	A simple system with one input and one output port.	28
3.2	Arrival and service curves.	29
3.3	Network calculus example.	31
3.4	A distributed system.	33
3.5	Model used by Trajectory approach.	34
3.6	AFDX network for Trajectory approach example.	36
3.7	Identification of busy periods.	37
3.8	Maximizing the arrival time in last node.	39
3.9	Schematic view of sample AFDX network for NuSMV model.	44
3.10	NuSMV code for modeling a VL.	45

3.11 Modeling of VLs at the End System and Switch.	46
3.12 NuSMV code for timers.	47
3.13 Output of NuSMV model checker.	48
3.14 Example of AFDX Network configuration.	51
3.15 Offsets between VLs of one ES.	52
3.16 Timed Automata for an end system.	53
3.17 Asynchronous behavior of ESs.	53
3.18 Timed Automata for switch output port.	54
3.19 Functions used for FIFO queue.	54
3.20 Timed Automata for Measuring VL.	55
3.21 Packet arrival instances and how a delay at output port appears as jitter.	56
3.22 Serialization of VLs after passing through an output port.	57
3.23 Timed Automata for Serialized VL.	57
3.24 Worst case scenario for VL1.	57
4.1 Illustration of a worst-case scenario	63
4.2 Worst-case for a frame x.	65
4.3 AFDX Network architecture for improved timed automata.	67
4.4 Sequence of $e1$	68
4.5 Timed automata of $e1$	69
4.6 Timed automata of $e2$	69
4.7 Timed automata of sporadic VL	70

4.8	Timed automata of $e9$	71
4.9	Timed automata of $SW1$	71
4.10	Timed automata of $SW3$	73
4.11	Timed automata of $SW5$	74
4.12	TA for the measurement	74
4.13	TA for integer clocks	76
4.14	A worst case scenario for $v1$	78
4.15	A simple timed automata.	80
4.16	Partial zone graph of the simple timed automata.	80
5.1	AFDX Network architecture.	85
5.2	Property 2.	88
5.3	Merging packet sequences and backlog calculation.	90
5.4	A simple AFDX Network of a small aircraft.	91
5.5	Sequences generated at input of Switch $S2$	92
5.6	Construction of sequences at output of Switch $S2$ Port 1.	93
5.7	Sequences generated at input of Switch $S1$	94
5.8	Construction of sequences at output of Switch $S1$ Port 1.	95
5.9	Construction of sequences at output of Switch $S3$ Port 1.	96
5.10	Network for reachable end to end delay illustration.	99
5.11	Pessimism of computed upper bounds.	100
5.12	Medium sized AFDX network	100

5.13 Periodic vs Sporadic traffic	101
5.14 Sporadic traffic in hyper period	102
5.15 Order and size of packet in switch output port	103
5.16 Idle time and queuing delay at a switch output port.	103
5.17 Idle time due to leaving VLs and its impact on packet under study.	105
5.18 Idle time and queuing delay at a switch output port.	106
6.1 AFDX Network of Airbus A380 aircraft.	114
6.2 Single VL and its paths with equivalent tree.	115
6.3 Connected Component of a graph which is equivalent to directly and indirectly interfering VLs.	116
6.4 VL10151 in isolation.	117
6.5 VL10151 directly linked paths.	118
6.6 VL10151 all linked paths.	119
6.7 Software architecture.	121
6.8 Flows of AFDX network of case study.	122
6.9 Results of the case study compared to Trajectory approach.	125
6.10 Pessimism in Trajectory approach.	125
6.11 Results of the case study compared to Network Calculus.	126
6.12 Pessimism in Network Calculus.	126
A.1 Comparison of Model Checking tools.	136
B.1 Software architecture.	142

B.2 JPPF Grid architecture.	144
-------------------------------------	-----

List of Tables

3.1	Exact worst-case delays.	58
4.1	AFDX network configuration data for improved timed automata	67
4.2	Synchronization among different groups of VLs.	75
5.1	AFDX network configuration data	84
5.2	End to End worst case delay for VL v3	97
5.3	End to End worst case delay under approximation algorithm illustration.	99
5.4	Performance comparison of algorithm.	99
5.5	Configuration example for Algorithm 2.	110
6.1	AFDX network configuration: BAGs and packet sizes	123
6.2	AFDX network configuration: crossed switches number of paths	123

Introduction

Contents

1.1	The Context	2
1.2	Contribution	4

In this era of modern science and technology, we rely heavily on the correct functioning of many Information and Communication Technology (ICT) systems. This trend is on the rise and while these systems are becoming more and more complex, at the same time they are massively encroaching on our daily life via the Internet and all kinds of embedded systems such as smart cards, hand-held computers, mobile phones, and high-end television sets. It is estimated that we are confronted with about 25 ICT devices on a daily basis [Baier 2008]. Many services such as electronic banking, on-line shopping (e-commerce) and smart card transactions are part of our routine life. The Internet alone accounts for about 10^{12} million US dollars cash flow. Modern transportation systems such as cars, trains and airplanes spend about one fourth of their production costs in ICT systems. ICT systems have become universal and omnipresent. They play vital role in control of the stock exchange market, they are the heart of telephone switches, they constitute Internet technology, and they are crucial for several kinds of medical systems, transportation systems and manufacturing systems. Our heavy reliance on these embedded systems make them very important and their reliable and correct operation has become a prime priority. Not only we want a good performance in terms like response times and processing capacity, but also the absence of annoying errors is one of the major quality demands.

The correct behavior of ICT systems is vital not only for money and comfort but in many cases, also for our lives. We don't like when our phones does not work properly or when our electronic gadgets reacts unexpectedly and wrongly to our issued commands. These software and hardware errors do not threaten our lives, but may have substantial financial consequences for the manufacturer. Examples are known where incorrect systems have caused valuable money loss to companies. The bug in Intel's Pentium II floating-point division unit in the early nineties caused a loss of about 475 million US dollars to replace faulty processors, and severely damaged



Figure 1.1 – June 4, 1996; The Ariane-5 crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value.

Intel’s reputation as a reliable chip manufacturer. The software error in a baggage handling system postponed the opening of Denver’s airport for 9 months, at a loss of 1.1 million US dollar per day [Baier 2008].

Errors can be catastrophic too. Notorious examples in the past are the fatal defects in the control software of the Ariane-5 missile (figure 1.1), the Mars Pathfinder, and the airplanes of the Airbus family. Similarly software are also used for the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems, and storm surge barriers. Consequently, bugs in such software can have disastrous impacts. For example, a software flaw in the control part of the radiation therapy machine “Therac-25” caused the death of six cancer patients between 1985 and 1987 as they were exposed to an overdose of radiation [Baier 2008].

All these examples remind us that it is very pertinent for any system to verify its correct intended operation and behaviour, specially for those which involve human lives. In this thesis, we strive for verification of an important avionics communication network known as Avionics Full-Duplex Switched Ethernet (AFDX). We will determine the exact worst case end to end communication delays of AFDX network. The context of this problem and brief background is presented in next section.

1.1 The Context

All modern aircraft have complex suit of on-board electronic devices used for various purposes, commonly referred to as Avionics Systems. Avionics systems of an aircraft are used in a wide

variety of different applications such as flight control, instrumentation, navigation, communication etc. These avionics systems need to communicate between themselves and exchange data, hence building "Avionics networks". Over the years, demand for data exchange has risen rapidly and avionics networks have evolved from dedicated links to shared buses and from shared buses to switched networks such as AFDX. Avionics Full-Duplex Switched Ethernet (AFDX) [ARINC 664 2005] is an avionics data network for safety critical applications and hence requires a very strict verification of its correct functioning. One important aspect of this verification is the maximum end to end communication delay for different devices connected to the network.

In any communication network, there is an end-to-end communication delay which occurs from the source generating a given message to the destinations receiving that message. For each message, this delay is composed of different parts: the transmission delays on links, the switching delays, the waiting times in output buffers. Knowing these delays is crucial for the overall system safety and reliability. However, finding the exact worst case delay for a given message is still an open problem, since every possible scenario has to be considered, leading to an intractable computation on any industrial configuration. Typically, this situation occurs in the context of avionics. Existing approaches for the computation of an exact worst-case delay in the context of the AFDX are based on model checking [Adnan 2010b, Charara 2006a] using Timed Automata [Alur 1994]. They cannot cope with configuration with more than ten VLs.

Many work has been devoted to the estimation of the worst-case delay for each message. By using techniques such as simulation and testing, it is possible to observe the network under study over long periods of times, thus considering a subset of all possible scenarios. Such an approach has been proposed in [Scharbarg 2009] for avionics networks. It provides an interval for the delay for each message. However, the delay for a message can be out of the obtained interval, since the approach does not consider all the possible scenarios. Consequently, simulation and testing do not provide us with exact worst case delays.

Analytical methods such as network calculus [Charara 2006a, Cruz 1991a], [Cruz 1991b, Fraboul 2002a, Le Boudec 2001, Li 2010] and trajectory approach [Bauer 2009, Bauer 2010, Martin 2006a] are used to compute an upper bound on the maximum delay for each flow. They guarantee that the delay can never be more than the calculated upper bound. These computed bounds are used for network certification but are pessimistic and cause under utilization of the network. The exact worst case delay for each flow is somewhere between the maximum observed delay and the calculated upper bounds, as shown in figure 1.2. In [Bauer 2010], authors have done analysis of this pessimism by the computation of an under approximation of this delay and comparing it with the results of sure upper bounds calculated by Network Calculus and Trajectory approaches. In figure 1.2, the difference between exact maximum delay and under

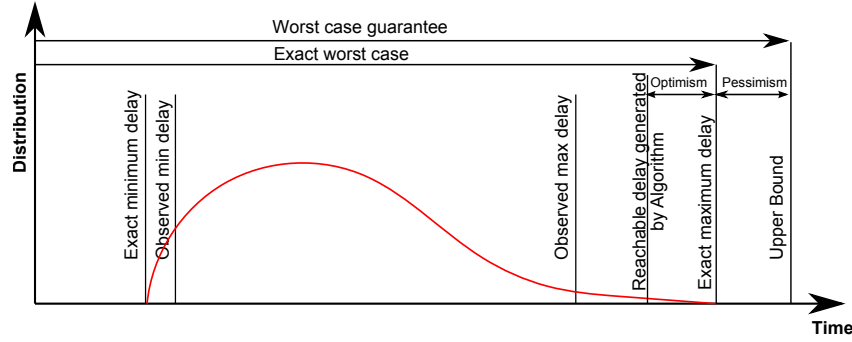


Figure 1.2 – AFDX network delays analysis.

approximation is the measure of optimism in algorithm used for under approximation while the difference between exact maximum delay and upper bound is the measure of pessimism in Network Calculus and Trajectory approaches used to calculate this bound. Without the knowledge of exact maximum delay, the difference between under approximation and upper bound is sum of optimism and pessimism between corresponding techniques. This gives us an estimation of pessimism in Network Calculus and Trajectory approaches.

Our aim, in this thesis, is to find exact worst case end to end communication delays. For this purpose we have explored existing tools and methodologies as well as developed new algorithms and tools where existing methods were not suitable for this task. The overview of this work is presented in the contribution section.

1.2 Contribution

The main objective of this thesis is to find exact worst case end to end communication delays of an AFDX network. For AFDX network, end to end communication delays can be approximated by using simulations, or they can be upper bounded by using analytical techniques such as Network Calculus or Trajectory approach. For computation of exact worst case delays, at present, we have only model checking approach but the model checking approach is limited only to small sized, proof of concept type networks and cannot analyze real life large sized industrial networks. These methods will be discussed in more detail in Chapter 3. Then, the problem is how to find exact worst case communication delays of large AFDX networks. Or, in other words, how can we improve models so that we are able to handle large networks. Also, another objective is to add local scheduling at the end systems in the computations which existing model checking approach doesn't incorporate.

Starting with the first model checking approach in [Charara 2006b], this approach can be improved: instead of analyzing the whole AFDX network simultaneously, only a part of the network can be considered and "divide and conquer" method can be used. This have been done in [Adnan 2010a] and presented in [Adnan 2010b, Adnan 2010c]. The idea is to consider one output port at a time and compute worst case delay at the port under study. This approach is successful in handling larger networks than the existing approach but it does not compute exact worst case delays for every scenario. Due to the port by port analysis approach, it is optimistic in certain cases where worst case delay on one port does not lead to overall end to end worst case delay. This is discussed in further detail in Chapter 3.

In this PhD Thesis, we propose to improve the Timed Automata models in two directions. First, scheduling of VLs at a local end system using offsets and asynchronous behaviour among all end systems have been added into the model, making the model of the AFDX network more realistic. Secondly, to make models efficient in resource (memory + computation) usage and to reduce search space, in order to cope with larger AFDX networks. In this context, we exploit AFDX network properties to reduce search space by considering only those cases which can be candidate for the worst case end to end delay. A first implementation considering this search space reduction has been presented in [Adnan 2011a, Adnan 2012]. A general purpose model checker, UPPAAL, is used for the developed timed automata models.

The models are detailed enough to capture periodic and sporadic flows with any BAG values. They also support local scheduling at each end system by using offsets. AFDX network with upto 32 VLs can be handled with this approach. This work is presented in more detail in Chapter 4.

At this point, we were still not able to compute worst case end to end delays of a real life industrial scale AFDX network by model checking. This is due to inherent exponential increase of state space in any model checking approach. But we were convinced that use of a general purpose model checker is also hampering our efforts to analyze larger networks. Therefore, we decided to use an approach which is more suited to AFDX network analysis. For this purpose, we started to develop a tool from scratch, which will allow us to exhaustively check all the cases which can be candidate for the worst case end to end delays in the AFDX network. This tool uses the same methodology and algorithms of state space reduction as used in Timed Automata based modeling approach discussed before. This work was presented in [Adnan 2011b]. The results are much better as compared to Timed Automata approach using UPPAAL software. We are able to compute end to end communication delays of a network which is twice the size of what Timed Automata based approach can handle. This approach is discussed in detail in Chapter 5.

We continued to pursue our main goal to analyze an industrial size AFDX network. With home made tool, we are at liberty to modify and change the software as required. We developed some algorithms and exploited more properties of AFDX network in order to reduce the search space to an extent where we were able to analyze industrial configuration of the AFDX network. We used this tool to study the end to end communication delays of a real life AFDX network, used on Airbus A380 aircraft. We are now able to compute end to end communication delays of an industrial sized network with about 1000 VLs having 6412 individual paths and more than 100 end systems. We are able to analyze all the VLs of this industrial AFDX network but not all the paths. We can analyze more than 60% paths. The reduction of state space is discussed in Chapter 5.5 and the case study of Airbus A380 network is presented in Chapter 6.

Our contribution in this thesis is to be able to find exact worst case end to end communication delays for large industrial size AFDX network and compare it with existing results to evaluate real pessimism of corresponding approaches.

Background: System Verification and AFDX Network

Contents

2.1	System Verification	7
2.1.1	Software Verification	10
2.1.2	Hardware Verification	10
2.1.3	Behavioral Verification	11
2.2	Model Checking	12
2.2.1	Formal Methods	12
2.2.2	Model-based Verification	13
2.2.3	History of Model Checking	15
2.2.4	Application of Model Checking in Networks	15
2.2.5	Characteristics of Model Checking	15
2.3	AFDX Network	18
2.3.1	History of Aircraft Data Networks (ADN)	19
2.3.2	Overview of AFDX	19
2.3.3	Virtual Links (VL)	21

In this chapter we will discuss about system verification and different methodologies available for this purpose. We will also talk about AFDX network in detail; how it works and what are the main building blocks.

2.1 System Verification

The complexity of ICT systems has increased with the advancement in technology. They have evolved from standalone systems to distributed systems; connecting and interacting with several

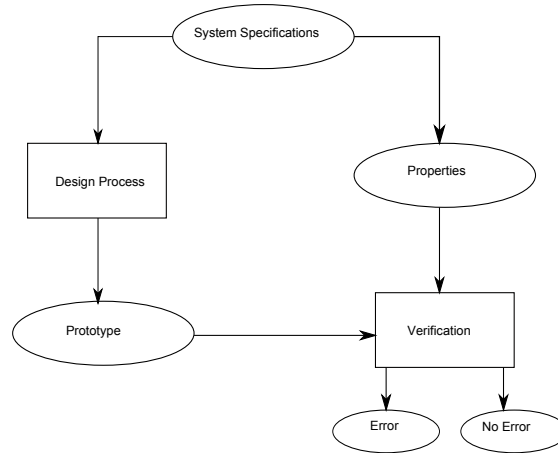


Figure 2.1 – Schematic view of system verification process.

other components and systems. This makes them more prone to errors as the probability and number of defects increases exponentially with the number of interacting system components. Increased complexity also makes it difficult for developers to debug and check systems for potential errors. In particular, phenomena such as concurrency and non-determinism that are central to modeling interacting systems turn out to be very hard to handle with standard techniques. Hence a lot of research is being carried out and efforts are being put in order to check Hardware and Software for correctness as well as compliance to it's specifications. These efforts are generally referred to as "System Verification" and is an active topic of research.

System verification techniques are integral part of all ICT system developments. Currently, the emphases of scientific community is on developing more reliable and accurate system verification techniques. In simple words, system verification is used to establish that the design or product under consideration satisfies certain properties. The properties to be validated are mostly obtained from the system's specification and can be quite elementary, e.g., to verify that the system should never be able to reach a situation in which no progress can be made (a deadlock scenario). The specifications describe what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system. A schematic view of verification process is depicted in figure 2.1.

Today's systems are very complex in it's nature and mostly comprise of interconnected sub systems. System verification is a vast field and can be further subdivided into major domains such as:

- Software verification
- Hardware verification
- Behavioral verification

There are different methodologies being used for system verification. These methodologies can be specific to one domain or applicable to more than one domain. These include:

- Measurements
- Tests
- Simulations
- Model Checking

In the context of this thesis, the verification of worst case end to end communication delays falls under behavioral verification domain. For this verification, all of the above mentioned methodologies can be used with varying degree of confidence or surety. Measurements are the quantitative indicators of the properties and performance criteria of the system under study. They can be very useful during the development phase or for troubleshooting. Tests are integral part of any system development. A system undergoes many tests from its inception to final product. At each state different tests are performed. Results of these tests dictates the progress to next level of development. Simulation is replication of a real world process or system over time. This replication should be as close to real world system as possible for better results. Simulations are used to investigate behaviour of the system and to validate proper functionality without using the actual system. Tests and simulations are quite similar in nature except that in tests, actual system is used. All of the above mentioned methodologies *i.e* measurements, tests and simulations only provide data at a particular instance under specific conditions, which means it does not verify system for all possible situations or scenarios. Therefore these methodologies can discover *many* anomalies in the system understudy but they can not verify that *all* the possible situations and scenarios have been covered. There is always a chance that a rare anomaly or event has not been tested. For covering all possible cases or scenarios, model checking is used. Model checking, for a given model of a system, exhaustively and automatically checks whether this model meets a given specification. Simulation can also check all possible cases or scenarios, and such simulation is referred to as *Exhaustive* Simulation. So, for 100% coverage of all possible scenarios, model checking and exhaustive simulation are the two methodologies which can be used. In the context of this thesis, for exact worst case end to end communication

delays, we will use both model checking and exhaustive simulation. These methodologies will be discussed in more detail in the next sections.

2.1.1 Software Verification

Software verification is a process of checking system's software for its compliance with specifications and expected requirements. Peer reviewing and testing are the major "software verification" techniques used in practice. A "peer review" refers to a software inspection carried out by a team of software engineers that preferably has not been involved in the development of software under review. The code of software is not executed but analyzed statically. Empirical studies indicate that peer review provides an effective technique that catches around 60% of the errors [Boehm 2001]. Despite its almost complete manual nature, peer review is thus a rather useful technique. Due to its static nature, experience has shown that subtle errors such as concurrency and algorithm defects are hard to catch using peer review.

"Software testing" is a significant part of any software engineering project [Whittaker 2000]. As opposed to peer review, which analyzes code statically without executing it, testing is a dynamic technique that actually runs the software. Testing takes the piece of software under consideration and provides its compiled code with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human beings. The main advantage of testing is that it can be applied to all sorts of software, ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically in-feasible; in practice only a small subset of these paths is treated. Testing can thus never be complete. That is to say, testing can only show the presence of errors, not their absence.

2.1.2 Hardware Verification

"Hardware Verification" is vital for preventing errors in hardware design. Hardware is subject to high fabrication costs; fixing defects after delivery to customers is difficult, and quality expectations are high. Whereas software defects can be repaired by providing users with patches or updates, hardware bug fixes after delivery to customers are very difficult and mostly require re-fabrication and redistribution. This has immense economic consequences. As mentioned

earlier, the replacement of the faulty Pentium II processors caused Intel a loss of about \$ 475 million. It is not surprising that chip manufacturers invest a lot in getting their designs right. Hardware verification is a well-established part of the design process. Emulation, simulation, and structural analysis are the major techniques used in hardware verification.

“Structural analysis” comprises several specific techniques such as synthesis, timing analysis, and equivalence checking. “Emulation” is a kind of testing. A re-configurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. As with software testing, emulation amounts to providing a set of stimuli to the circuit and comparing the generated output with the expected output as laid down in the chip specification. To fully test the circuit, all possible input combinations in every possible system state should be examined. This is impractical and the number of tests needs to be reduced significantly, yielding potential undiscovered errors. With “simulation”, a model of the circuit at hand is constructed and simulated. Models are typically provided using hardware description languages such as *Verilog* or *VHDL* that are both standardized by *IEEE*. Based on stimuli, execution paths of the chip model are examined using a simulator. These stimuli may be provided by a user, or by automated means such as a random generator. A mismatch between the simulator’s output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice.

2.1.3 Behavioral Verification

Behavior of a system refers to its expected outputs for a given set of assumed inputs. In simple words, Behavioral Verification is the process of verification of system’s “behavior” under given conditions. System’s behavior is combined effect of its software and hardware functioning. Even though a system is verified separately for its software and hardware, it is equally important to verify the system at more abstract and conceptual levels. For example, communication protocols, compliance of specified rules, and interaction among subsystems must be verified before starting development of hardware and software for each individual subsystem. Behavioral verification requires a “model” of the system. This model is a formal way of describing the system over which we can use certain queries and properties to verify its behavior. Different tools incorporating various techniques exist which help in modeling a system for its behavioral verification such as TINA, NuSMV, SPIN, UPPAAL etc. A comprehensive list of such tools can be consulted in [Appendix A](#).

It must be noted that none of the software and hardware verification techniques described earlier gives us the 100% confidence about system correctness due to the same limitation of unreasonably large set of possible scenarios. If we need absolute surety of our design, then we must find a way to test all possible scenarios and that's where model checking helps us. Model checking approach searches all possible scenario exhaustively to prove correctness of the model under test. An important aspect of model checking is that it's as good as the model itself, i.e we must ensure that the model correctly represents the system before we start the model checking. In the following sections, basic theory of model checking is presented. The details of available model checking software, called "model checkers", and their application to AFDX network is described in Chapter 3.2 and Chapter 4 respectively. Further discussion about model checking is presented in Appendix A.

2.2 Model Checking

In general more time and effort are spent on verification than on construction in software and hardware design of complex systems. Naturally, many techniques are sought to reduce and ease the verification efforts while increasing their coverage. One such technique is the use of "Formal methods" which is known to offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time.

2.2.1 Formal Methods

To put it in a nutshell, formal methods can be considered as "the applied mathematics for modeling and analyzing ICT systems". Their aim is to establish system correctness with mathematical rigor. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex software and hardware systems. Besides, formal methods are one of the "highly recommended" verification techniques for software development of safety critical systems according to, e.g., the best practices standard of the IEC (International Electro-technical Commission) and standards of the ESA (European Space Agency). The resulting report [Baier 2008] of an investigation by the FAA (Federal Aviation Authority) and NASA (National Aeronautics and Space Administration) about the use of formal methods concludes that "Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers."

During the last two decades, research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. These techniques are accompanied by powerful software tools that can be used to automate various verification steps. Investigations have shown that formal verification procedures would have revealed the exposed defects in, e.g., the Ariane-5 missile, Mars Pathfinder, Intel's Pentium II processor, and the Therac-25 therapy radiation machine.

2.2.2 Model-based Verification

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner. It turns out that prior to any form of verification, the accurate modeling of systems often leads to the discovery of incompleteness, ambiguities, and inconsistencies in informal system specifications. Such problems are usually only discovered at a much later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). Due to unremitting improvements of underlying algorithms and data structures, together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples are nowadays applicable to realistic designs. As the starting point of these techniques is a model of the system under consideration, we have as a given fact that any verification using model-based techniques is only as good as the model of the system.

Model checking is a verification technique that explores all possible system states, commonly known as state-space, in a brute-force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories. State-of-the-art model checkers can handle state spaces of about 10^8 to 10^9 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} up to even 10^{476} states) can be handled for specific problems [Straunstrup 2000]. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

Typical properties that can be checked using model checking are of a qualitative nature: Is

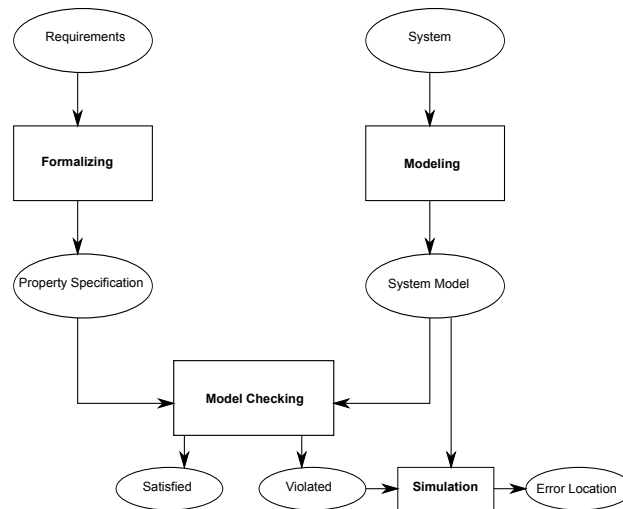


Figure 2.2 – Schematic view of Model Checking process.

the generated result OK?, Can the system reach a deadlock situation? e.g., when two concurrent programs are waiting for each other and thus halting the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or, Is a response always received within 8 minutes? Model checking requires a precise and unambiguous statement of the properties to be examined. As with making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation. For instance, the formalization of all system properties for a subset of the ISDN user part protocol revealed that 55% (!) of the original, informal system requirements were inconsistent [Holzmann. 1994]. The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming languages like C or Java or hardware description languages such as Verilog or VHDL. Note that the property specification prescribes what the system should do, and what it should not do, whereas the model description addresses how the system behaves. The model checker examines all relevant system states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model (or the property) accordingly (see Figure 2.2).

2.2.3 History of Model Checking

Model checking originates from the independent work of two pairs in the early eighties: Clarke and Emerson [Clarke 1981] and Queille and Sifakis [Queille 1982]. The term model checking was coined by Clarke and Emerson. The brute-force examination of the entire state space in model checking can be considered as an extension of automated protocol validation techniques by Hajek [Hajek 1978] and West [West 1978, West 1989]. While these earlier techniques were restricted to checking the absence of deadlocks or livelocks, model checking allows for the examination of broader classes of properties. Introductory papers on model checking can be found in [Clarke 1996a, Clarke 2000, Clarke 1996b, Merz 2001, Wolper 1995]. The limitations of model checking were discussed by Apt and Kozen [Apt 1986]. More information on model checking is available in the earlier books by Holzmann [Holzmann 1990], McMillan [McMillan 1993], and Kurshan [Kurshan 1994] and the more recent works by Clarke, Grumberg, and Peled [Clarke 1999], Huth and Ryan [Huth 1999], Schneider [Schneider 2004], and Bérard et al. [Bérard 2001]. Automated analysis of designs, in particular verification by model checking, has recently been described by Ruys and Brinksma in [Ruys 2003].

2.2.4 Application of Model Checking in Networks

Model checking has been used for verification of different systems in the past. In the domain of networks, it has been used to verify redundant media extension of Ethernet PowerLink [Steve 2007]. It has also been used in networked automation systems [Ruel 2008] and in functional analysis of real-time protocol in an networked control system [Fidge 2006]. For Integrated Modular Avionics (IMA) [ARINC 653 1997], the bounds on end to end functional delays have been studied in [Lauer 2010]. All these applications of model checking are different than what we do in this thesis. None of the above approaches find exact worst case communication delays over the network. Most of these approaches use either an abstraction of the network with basic functionality such as *NetworkOK*, *NetworkCongested*, *OnTime*, *TooLate* as in [Fidge 2006] or they use upper bounds of network communication delays calculated by Network Calculus or Trajectory approach as in [Lauer 2010].

2.2.5 Characteristics of Model Checking

Model Checking can be defined as “an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given

state in) that model." The next section briefly explains the general process of model checking followed by its advantages, limitations and role in system development cycle.

2.2.5.1 Model Checking Process

Model checking process can be divided in following different phases:

- Modeling phase:
 - model the system under consideration using the model description language of the model checker at hand;
 - as a first sanity check and quick assessment of the model perform some simulations;
 - formalize the property to be checked using the property specification language.
- Running phase: run the model checker to check the validity of the property in the system model.
- Analysis phase:
 - is property satisfied? If yes then check next property (if any);
 - is property violated? If yes then system did not respect its specification. Therefore:
 1. analyze generated counterexample by simulation;
 2. refine the model, design, or property;
 3. repeat the entire procedure.
 - out of memory? If yes then try to revise the abstraction level of the model to reduce its size and try again.

2.2.5.2 Strengths and Weaknesses of Model checking

Following are the main strengths of model checking approach:

- It is a general verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports partial verification, i.e., properties can be checked individually, thus allowing focus on the essential properties first. No complete requirement specification is needed.

- It is not vulnerable to the likelihood that an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.
- It provides diagnostic information in case a property is invalidated; this is very useful for debugging purposes.
- It is a potential “push-button” technology; once the model has been developed, the use of model checking tools requires neither a high degree of user interaction nor a high degree of expertise.
- It enjoys a rapidly increasing interest by industry; several hardware companies have started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers have become available.
- It can be easily integrated in existing development cycles; its learning curve is not very steep, and empirical studies indicate that it may lead to shorter development times.
- It has a sound and mathematical underpinning; it is based on theory of graph algorithms, data structures, and logic.

Following are the weaknesses of model checking:

- It is mainly appropriate to control-intensive applications and less suited for data intensive applications as data typically ranges over infinite domains.
- Its applicability is subject to decidability issues; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.
- It verifies a system model, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques, such as testing, are needed to find fabrication faults (for hardware) or coding errors (for software). This highly depends on the level of abstraction in the model of the system.
- It checks only stated requirements, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the state-space explosion problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem, models of realistic systems may still be too large to fit in memory.

- Its usage requires some expertise in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.
- It is not guaranteed to yield correct results: as with any tool, a model checker may contain software defects.
- It does not allow checking generalizations: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

Model checking has great potential in system verification and removing rare to find bugs. It can formally verify properties of the system, and it can be used to check correct behavior of the system. Model checking is the choice when we want a complete verification of the system which simulations can not provide. Model checking considers all possible cases, and hence we can apply it to AFDX network in order to find exact end to end communication delays. We will discuss the application of this technique to find exact end to end communication delays of AFDX network [ARINC 664 2005] in coming sections (Chapter 3.2.1, Chapter 4), but first let's describe AFDX Network.

2.3 AFDX Network

All modern aircraft have complex suit of on-board electronic devices used for various purposes, commonly referred to as Avionics Systems. Avionics systems of an aircraft are used in a wide variety of different applications such as flight control, instrumentation, navigation, communication etc. These avionics systems need to communicate between themselves and exchange data, hence building "Avionics networks". Over the years, demand for data exchange has risen rapidly and avionics networks have evolved from dedicated links to shared buses to switched networks such as AFDX. Avionics Full-Duplex Switched Ethernet (AFDX) [ARINC 664 2005] is a data network for safety critical applications that utilizes dedicated bandwidth while providing deterministic Quality of Service (QoS). AFDX is based on IEEE 802.3 Ethernet technology and utilizes commercial off-the-shelf (COTS) components with certain constraints applied. It is described specifically by Part 7 of the ARINC 664 Specification, as a special case of a profiled version of an IEEE 802.3 network per parts 1 & 2, which defines how Commercial Off-the-Shelf networking components will be used for future generation Aircraft Data Networks (ADN). The six primary aspects of AFDX include full duplex, redundancy, deterministic, high speed performance, switched and profiled network.

2.3.1 History of Aircraft Data Networks (ADN)

Prior to AFDX, Aircraft Data Networks utilized primarily the ARINC 429 standard. This standard, developed over thirty years ago and still widely used today, has proven to be highly reliable in safety critical applications. This ADN can be found on a variety of aircraft from both Boeing and Airbus, including the B737, B747, B757, B767 and Airbus A330, A340, A380 and the upcoming A350. ARINC 429 utilizes a unidirectional bus with a single transmitter and up to twenty receivers. A data word consists of 32 bits communicated over a twisted pair cable using the Bipolar Return-to-Zero Modulation. There are two speeds of transmission: high speed operates at 100 kbit/s and low speed operates at 12.5 kbit/s. ARINC 429 operates in such a way that its single transmitter communicates in a point-to-point connection, thus requiring a significant amount of wiring which amounts to added weight.

Another standard, ARINC 629, introduced by Boeing for the 777 provides increased data speeds of up to 2 Mbit/s and allowing a maximum of 120 data terminals. This ADN operates without the use of a bus controller thereby increasing the reliability of the network architecture. The drawback of this system is that it requires custom hardware which can add significant cost to the aircraft. Because of this, other manufacturers did not openly accept the ARINC 629 standard.

ARINC 664 is defined as the next-generation aircraft data network (ADN). It is based upon IEEE 802.3 Ethernet and utilizes commercial off the shelf hardware thereby reducing costs and development time. AFDX builds on this standard, as is formally defined in Part 7 of the ARINC 664 specification. AFDX was developed by Airbus Industries for the A380. It has since been accepted by Boeing and is used on the Boeing 787 Dreamliner. AFDX bridges the gap on reliability of guaranteed bandwidth from the original ARINC 664 standard. It utilizes a cascaded star topology network, where each switch can be bridged together to other switches on the network. By utilizing this form of network structure, AFDX is able to significantly reduce wire runs thus reducing overall aircraft weight. Additionally, AFDX provides dual link redundancy and Quality of Service (QoS). Figure 2.3 compares basic architecture of ARINC and AFDX networks.

2.3.2 Overview of AFDX

AFDX adopted concepts (token bucket) from the telecom standard, Asynchronous Transfer Mode (ATM), to fix the shortcomings of IEEE 802.3 Ethernet such as in-deterministic behavior of "Carrier sense multiple access with collision detection (CSMA/CD)". By adding key elements

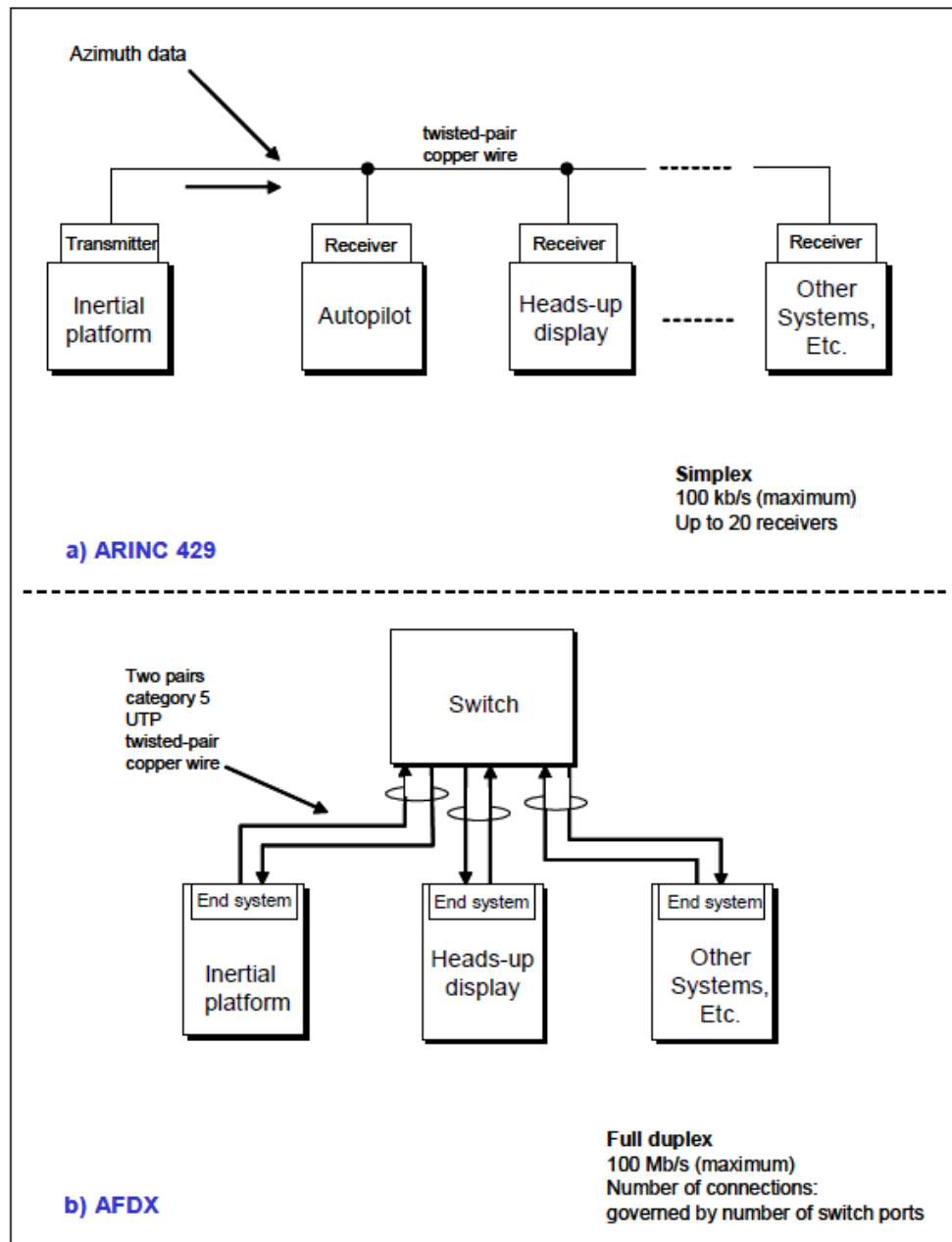


Figure 2.3 – ARINC 429 vs AFDX architecture (courtesy condor engineering inc.)

from Asynchronous Transfer Mode (ATM) to those already found in Ethernet, and constraining the specification of various options, a highly reliable Full-Duplex deterministic network is created providing guaranteed bandwidth and Quality of Service. Through the use of Full-Duplex Ethernet, the possibility of transmission collisions is eliminated. A highly intelligent switch, common to the AFDX network, is able to buffer transmission and reception packets. Through the use of twisted pair or fiber optic cables, Full-Duplex Ethernet uses two separate pairs or strands for transmit and receiving data. AFDX extends standard Ethernet to provide high data integrity and deterministic timing. Further a redundant pair of networks is used to improve the system integrity. Figure 2.4 depicts a generic AFDX network. It specifies inter-operable functional elements at the following OSI Reference Model layers:

- Data Link (MAC and Virtual Link addressing concept);
- Network (IP and ICMP);
- Transport (UDP and optionally TCP)
- Application (Network) (Sampling, Queuing, SAP, TFTP and SNMP).

The main elements of an AFDX network are:

- AFDX End Systems
- AFDX Switches
- AFDX Links

2.3.3 Virtual Links (VL)

The central feature of an AFDX network are its Virtual Links (VL). In one abstraction, it is possible to visualize the VLs as an ARINC 429 style network each with one source and one or more destinations as shown in figure 2.5. Virtual Links are unidirectional logic path from the source end-system to all of the destination end-systems. Unlike that of a traditional Ethernet switch which switches frames based on the Ethernet destination or MAC address, AFDX routes packets using a Virtual Link ID. The Virtual Link ID is a 16-bit Unsigned integer value that follows the constant 32-bit field. The switches are designed to route an incoming frame from one, and only one, End System to a predetermined set of End Systems. There can be one or more receiving End Systems connected within each Virtual Link. Each Virtual Link is allocated

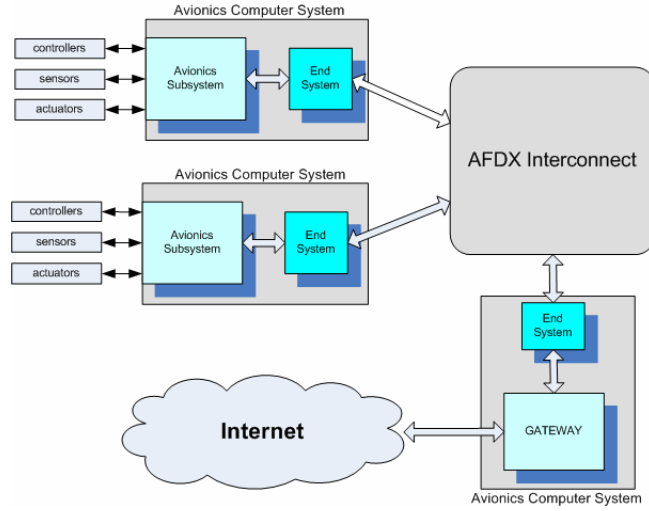


Figure 2.4 – AFDX network (courtesy condor engineering inc.)

dedicated bandwidth known as Bandwidth Allocation Gap (BAG) with the total amount of bandwidth defined by the system integrator. However total bandwidth can not exceed the maximum available bandwidth on the network. Bi directional communications must therefore require the specification of a complimentary VL. Each VL is frozen in specification to ensure that the network has a designed maximum traffic, hence performance. Also the switch, having a VL configuration table loaded, can reject any erroneous data transmission that may otherwise swamp other branches of the network. Additionally, there can be sub-virtual links (sub-VLs) that are designed to carry less critical data. Sub-virtual links are assigned to a particular Virtual Link. Data is read in a round robin sequence among the Virtual Links with data to transmit. Also sub-virtual links do not provide guaranteed bandwidth or latency due to the buffering, but AFDX specifies that latency is measured from the traffic regulator function anyway.

A generic AFDX switch architecture is shown in figure 2.6. Each switch has filtering, policing, and forwarding functions that should be able to process at least 4096 VLs (this seems like a system specific derived requirement in part 7). Therefore, in a network with multiple switches (cascaded star topology), the total number of Virtual Links is nearly limitless. There is no specified limit to the number of Virtual Links that can be handled by each End System (except the one imposed by the VL ID field size in the packet header), although this will be determined by the BAG rates and max frame size specified for each VL versus the Ethernet data rate. However, the number sub-VLs that may be created in a single Virtual Link is limited to four. The switch must also be non-blocking at the data rates that are specified by the system integrator, and in practise this may mean that the switch shall have a switching capacity that is the sum of all of its physical ports.

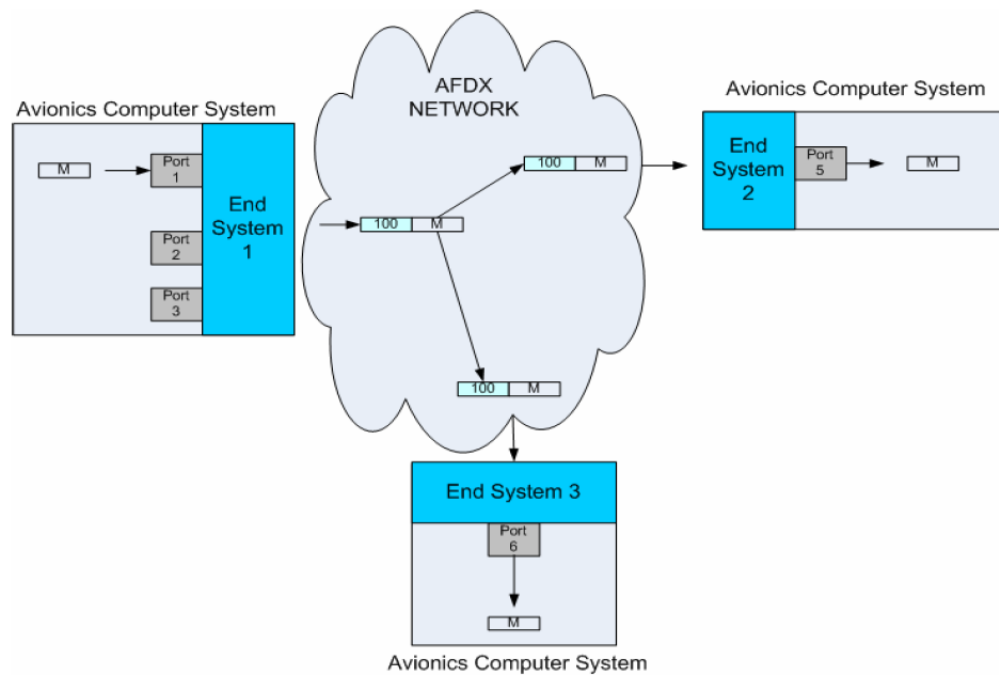


Figure 2.5 – AFDX virtual links.

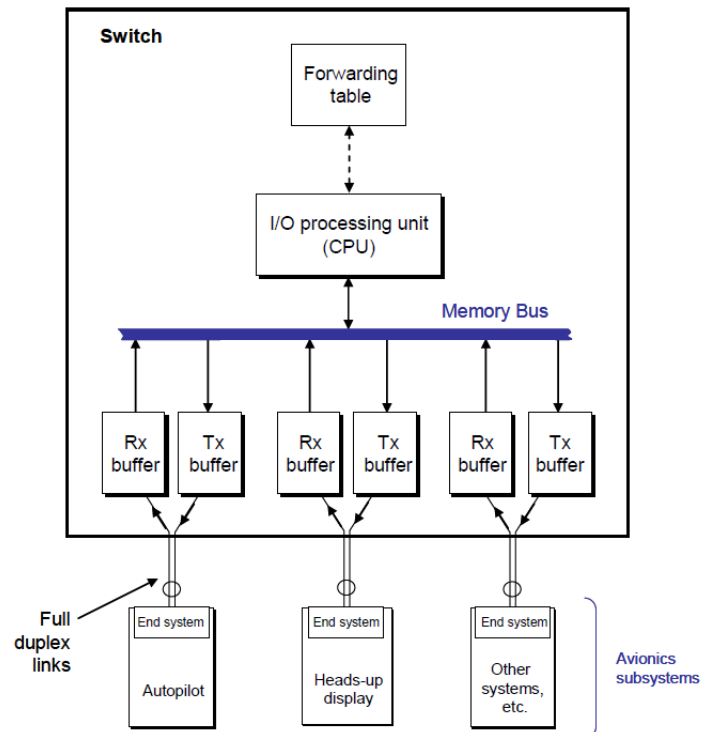


Figure 2.6 – AFDX switch architecture.

The AFDX network is being adapted in many modern aircraft. At present it is being used in Airbus A380, Boeing 787, Airbus A400M, Airbus A350, Sukhoi Superjet 100, AgustaWestland AW101, AgustaWestland AW149 and some others.

State of the Art: Methods to Compute the Worst Case End to End Delays in an AFDX Network

Contents

3.1	Bounds of Worst Case End-to-End Communication Delays	26
3.1.1	Network Calculus	26
3.1.2	Trajectory Approach	32
3.1.3	Pessimism of Network calculus and Trajectory approach	38
3.1.4	Conclusion	40
3.2	Exact Worst Case End-to-End Communication Delays	40
3.2.1	Model Checking	40
3.2.2	Exhaustive Simulation	58
3.2.3	Conclusion	58
3.3	Conclusion	59

For certification purpose, it is mandatory to prove maximum latency in the airline avionics network, such as AFDX. Indeed, with respect to real-time characteristics of avionics systems, communication infrastructure among avionics equipment need to guarantee that when one equipment sends a message to another equipment, the transmission delay does not exceed a maximum value. This transmission delays depends on different possible scenarios, *i.e.* the instant when each message is sent, the position of the message in the queue, etc. In order to find the maximum transmission delay, we need to compute the worst-case configuration which leads to this delay.

In this chapter we will present these techniques and methods which are being used for worst case delay analysis for an AFDX network. We can broadly categorize the techniques to find

worst case communication delays in AFDX network into two main categories:

- Techniques which give a sure upper bound on end to end communication delays.
 - Network calculus and Trajectory approach
- Techniques which evaluate exact end to end communication delays.
 - Model checking and Exhaustive simulation

At present, Network calculus is actually used for AFDX certification, because it was one of the first efforts in this direction, but other methods should be considered and will be presented in the following sections.

3.1 Bounds of Worst Case End-to-End Communication Delays

To guarantee the maximum transmission delays, a first approach consists in finding the bounds of the worst case end to end delays. Two methods can be applied in this context and they will be presented in this section: Network calculus and Trajectory approach.

3.1.1 Network Calculus

Network calculus (NC) is a theoretical framework that provides deep insights into flow problems encountered in networking. The foundation of network calculus lies in the mathematical theory of *dioids*, and in particular, the Min-Plus dioid (also called Min-Plus algebra) [[Jean-Yves Le Boudec 2001](#)]. Network calculus is extensively used for analyzing performance guarantees in computer networks. Network calculus was first applied to AFDX network by Christian Fraboul et al. in [[Fraboul 2002b](#), [Frances 2006](#)], which lead to the certification of AFDX Network on Airbus A380 aircraft. In following sections, network calculus theory and it's application to AFDX network is presented.

3.1.1.1 Network Calculus Theory

In this section a brief overview of network calculus is presented. A more detailed presentation can be found in [[Jean-Yves Le Boudec 2001](#)].

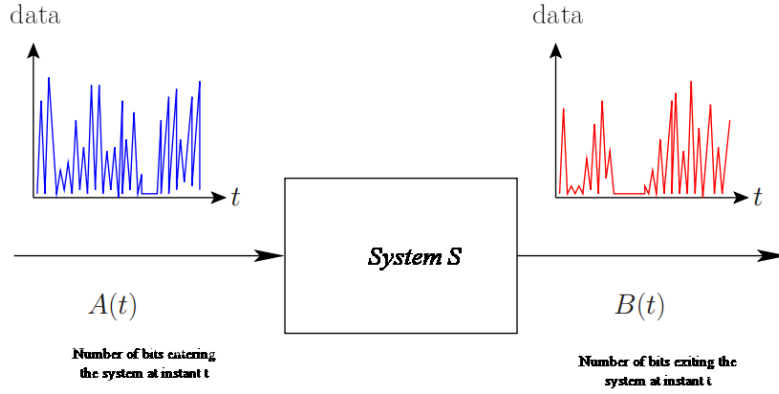
To provide guarantees to data flows in a network, it is necessary that the network has some kind of bounds on its resources. This means that all sources must have guarantees on maximum traffic emission and all service providing elements must have guarantees on capacity. To model the data generated by network elements (sources), with a given data rate constraint, a concept of arrival curve is used. In order to provide reservations, network nodes in return need to offer some guarantees to flows. This is done by packet schedulers. The details of packet scheduling are abstracted by using the concept of service curve. Below we describe these concepts of network calculus.

Consider a simple system S as shown in figure 3.1. It has one input and one output. Figure 3.1a shows the number of bits entering the system S at any time instant t and number of bits exiting the system S at any time t . Figure 3.1b shows the same system S but the input and output traffic is cumulative instead of being instantaneous. This means that the graphs display sum of number of bits received or sent till time t . We can see that with cumulative traffic, the graph is always increasing because of the summation of bits and is easier to understand. In network calculus, such graphs are best represented by the use of cumulative functions.

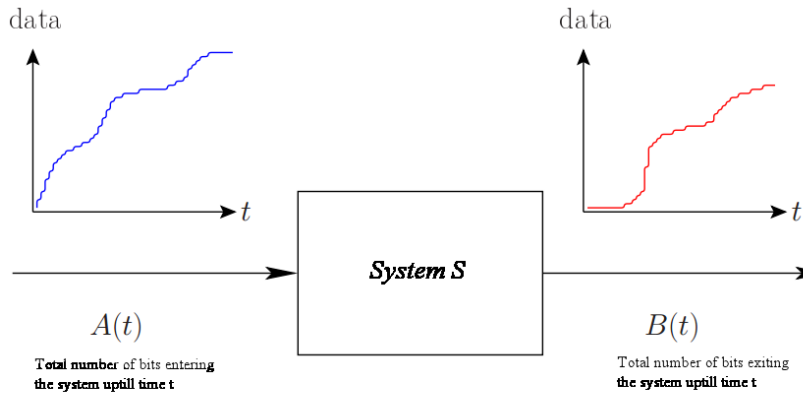
Cumulative Functions: Network calculus models data flows, as cumulative functions which can be both continuous time and discrete time. A cumulative function $R(t)$ is defined as the number of bits in the flow during time interval $[0, t]$. Function $R(t)$ is always a wide-sense increasing function. Generally it is assumed that $R(0)=0$, unless stated otherwise. In figure 3.1b, both the input traffic and output traffic is an example of cumulative function. Cumulative functions describe the relationship between total number of bits and time; they give us sum of all bits arrived (or left) till a time instant t .

Input and Output Function: Let's consider a system S as a black-box. S receives input data and after processing it transmits the data at output. If input is defined by cumulative function $R(t)$, then the output is defined by another cumulative function $R^*(t)$ called as *output function*. Figure 3.1c shows an example of input and output functions. The graph in black represents input function and graph in red represents corresponding output function. The horizontal distance $d(t)$ between input and output function graph represents the delay that an input traffic will experience while the vertical distance $x(t)$ between input and out function represents the total number of bits present in the system as backlog. $R_1(t)$ and $R_1^*(t)$ show a continuous function of continuous time (fluid model); we assume that packets arrive bit by bit, for a duration of one time unit per packet arrival. Functions $R_2(t)$ and $R_2^*(t)$ show continuous time with discontinuities at packet arrival times (times 1, 4, 8, 8.6 and 14); we assume here that packet arrivals are observed only when the packet has been fully received.

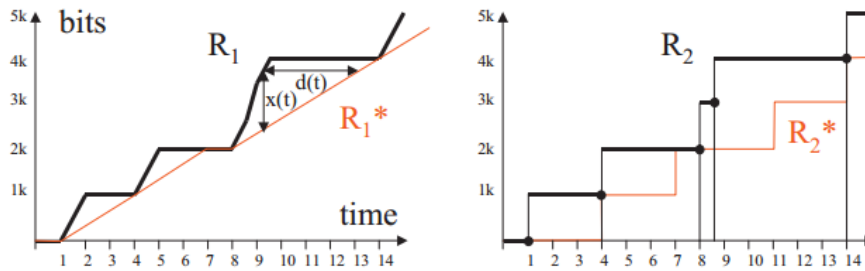
Arrival Curve: Arrival curve is a way to constrain data emitted by sources. Given a wide-sense increasing function α defined for $t \geq 0$, we say that a flow defined by cumulative function R is constrained by α if and only if for all $s \leq t$:



(a) Instantaneous traffic at any given instance t .



(b) Cumulative traffic till time t .



(c) Examples of Input Output cumulative functions.

Figure 3.1 – A simple system with one input and one output port.

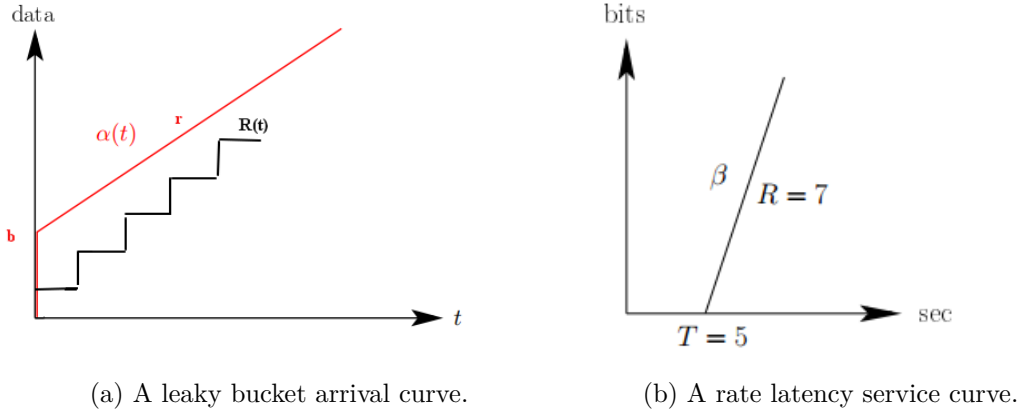


Figure 3.2 – Arrival and service curves.

$$R(t) - R(s) \leq \alpha(t - s) \quad (3.1)$$

The equation implies that α is an arrival curve for function R , or R is α smooth. In simple words, α is an upper bound on R . A well known example of arrival curve is a leaky bucket arrival curve $\gamma_{r,b}$ shown in figure 3.2a where b represents initial burst of data and r represents steady rate. It is defined as:

$$\gamma_{r,b}(t) = \begin{cases} 0 & \text{if } t < 0 \\ rt + b & \text{otherwise} \end{cases} \quad (3.2)$$

Service Curve: Consider a system S and a flow through S with input and output function R and R^* respectively. We say that S offers to the flow α a service curve β if and only if β is wide sense increasing, $\beta(0) = 0$ and $R^* \geq R \otimes \beta$, where \otimes is min-plus convolution operator. This definition implies that β is also a wide sense increasing function and for all $t \geq 0$ we have:

$$R^*(t) \geq \inf_{s \leq t} (R(s) + \beta(t - s)) \quad (3.3)$$

This means that the system S offers a minimum guaranteed service characterized by β to inputs. A well known example of service curve is rate latency service curve $\beta_{R,T}$ shown in figure 3.2b where R represents rate and T represents the bound on maximum initial delay for the bits of input flow. It is defined as:

$$\beta_{R,T}(t) = \begin{cases} 0 & \text{if } t < T \\ R(t - T) & \text{otherwise} \end{cases} \quad (3.4)$$

Network Calculus Bounds: Network calculus has three main results. These are bounds for

lossless systems with service guarantees. The first result is about backlog bound. It states that the vertical distance between arrival curve and service curve presents upper bound on backlog. More precisely, if a flow R constrained by arrival curve α traverses a system that offers a service curve β then the backlog $R(t) - R^*(t)$ (the difference between input and output function) for all values of time t satisfies the following:

$$R(t) - R^*(t) \leq \sup_{s \geq 0} \{\alpha(s) - \beta(s)\} \quad (3.5)$$

The second result is about delay bound. It states that the horizontal distance between the arrival curve and service curve presents upper bound on the delay experienced by the traffic. More precisely, if a flow R constrained by arrival curve α traverses a system that offers a service curve β then the delay d experienced by the flow R is bounded by the maximum horizontal distance between the curve α and β (denoted by $h(\alpha, \beta)$). The delay $d(t)$ for all values of time t satisfies the following:

$$d(t) \leq h(\alpha, \beta) = \sup_{z \geq 0} \{\beta^{-1}(z) - \alpha^{-1}(z)\} \quad (3.6)$$

The third result is about output flow. It states that the output flow of a system can be constrained with an arrival curve α^* , obtained by min-plus deconvolution (\oslash) of arrival curve α and service curve β . *i.e.*:

$$\alpha^* = \alpha \oslash \beta \quad (3.7)$$

Concatenation: Another important result of network calculus is about concatenation of nodes. It states that if a flow passes from two or more systems in sequence, then we can merge these systems into a single system. More precisely, assume a flow traverses systems $S1$ and $S2$ in sequence. Assume that $S1$ offers a service curve β_1 and $S2$ offers a service curve β_2 to the flow. Then the concatenation of the two systems offers a service curve of $\beta_1 \otimes \beta_2$ to the flow.

3.1.1.2 Application to AFDX

To use network calculus on AFDX network, the traffic must respect some constraints. As discussed in 2.3, a virtual link (VL) is a static mono-sender multicast flow. A VL is constrained by minimum frame size S_{min} , maximum frame size S_{max} and a minimum interval between two consecutive frames called BAG . Therefore, a VL can be modeled in network calculus as a leaky bucket arrival curve $\gamma_{\frac{S_{max}}{BAG}, S_{max}}$. Similarly, a switch output port can be modeled as rate latency service curve $\beta_{R,T}$ where R is the throughput of Ethernet link and T is the switching latency

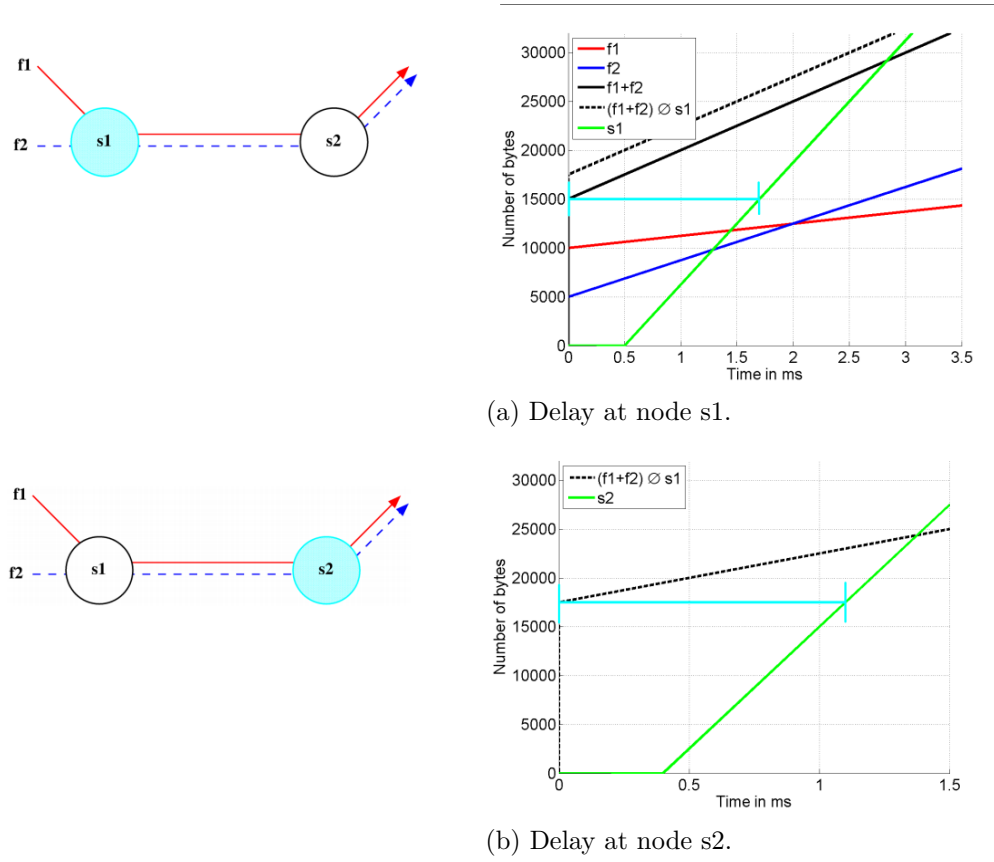


Figure 3.3 – Network calculus example.

of the AFDX switch. The results can be propagated through the entire network by using the output flow equation 3.7. The output of one switch becomes input of the next switch. This approach was applied to industrial configuration of the AFDX network by Christian Fraboul et al. in [Fraboul 2002b, Frances 2006]. In order to explain how the network calculus can be used to find communication delays, we will consider a simple example.

Network Calculus Example: Let's consider a simple example where two flows $f1$ and $f2$ pass through two nodes $s1$ and $s2$ as shown in figure 3.3. Assuming that flows are represented by leaky bucket arrival curve and nodes offer a rate latency service curve, we can find the delay experienced by flow $f1$ in both nodes $s1$ and $s2$ graphically, as shown in figure 3.3. At node $s1$, there are two flows entering the node. We can aggregate these flows to a single flow which is equivalent to $f1 + f2$ whose arrival curve is shown as black line in figure 3.3a. This arrival curve is deconvoluted with service curve of node $s1$ (shown as green line) to get the arrival curve at the output of node $s1$ (dotted black line). The bound on maximum delay at this node is the maximum horizontal distance between arrival curve of $f1 + f2$ (black line) and service curve of

s_1 (green line). The output of node s_1 is input of node s_2 . Therefore at node s_2 , output arrival curve of node s_1 (dotted black line) is taken as input arrival curve, and the service curve for node s_2 is used for calculations at this node, as shown in figure 3.3b. The bound on maximum delay at this node is the maximum horizontal distance between arrival curve (dotted black line) and service curve of s_2 (green line). The end to end delay for flow f_1 is sum of delays at node s_1 and s_2 .

This worst case delay analysis is obviously pessimistic. The Network Calculus is a holistic approach[Tindell 1994] and the worst case scenario is considered on each node visited by a flow, taking into account maximum possible jitters introduced by previously visited nodes. This approach can indeed lead to impossible scenarios. There are also other pessimism causes, intrinsic to the Network Calculus theory, as envelopes are used instead of the exact arrival curve and service curves.

3.1.2 Trajectory Approach

The Trajectory approach has been developed to get deterministic upper bounds on end-to-end response times in distributed systems [Martin 2004, Martin 2006a, Martin 2006b, Migge 1999]. This approach considers a set of sporadic flows with no assumption concerning the arrival times of packets. Thus the obtained upper bounds are valid for every possible arrival times of packets. Trajectory approach considers the sequence of nodes visited by a frame along its trajectory. Unlike the holistic approach in network calculus, the Trajectory approach is based on the analysis of the worst case scenario experienced by a packet on its trajectory and not on any visited node. This timing analysis approach enables to focus on a packet from a given flow, and to construct the packet sequences in each crossed node. The resulting jitters and delays lead to an end-to-end communication delay computation, which can then be compared to the upper bounds obtained by deterministic Network Calculus approach.

3.1.2.1 Trajectory Theory

In order to explain the theory behind Trajectory approach, we will consider a simple example. Suppose the architecture of a distributed system depicted in figure 3.4 [Martin 2006a]. Such a system is composed of a set of processing nodes (seven in figure 3.4) with some links between them. Each flow crossing this system follow a static path which is an ordered sequence of nodes. In the example of figure 3.4, there are two flows τ_1 and τ_2 . τ_1 follows the path $P_1 = \{4,5,6,7\}$. Node 4 is the entry point of flow τ_1 in this system, and is often referred to as *ingress node*. The

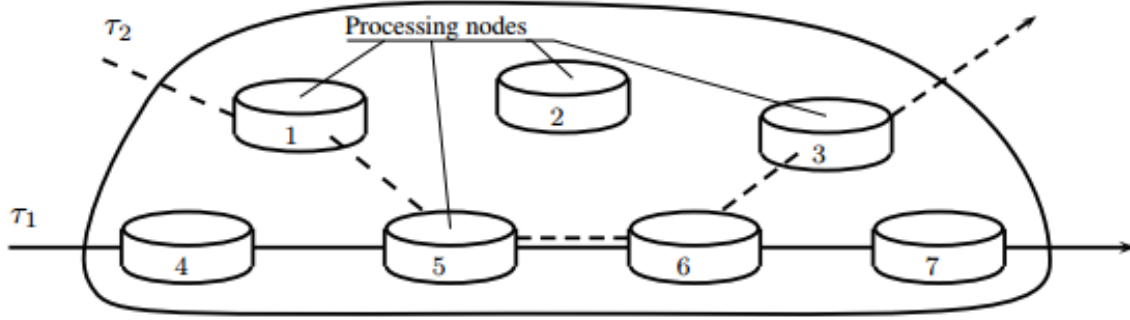


Figure 3.4 – A distributed system.

Trajectory approach assumes, with regards to any flow τ_i following path P_i , that any flow τ_j following path P_j , with $P_j \neq P_i$ and $P_j \cap P_i \neq \emptyset$, never visits a node of path P_i after having left this path. In the example of figure 3.4, $P_2 = \{1, 5, 6, 3\}$ and $P_1 \cap P_2 = \{5, 6\}$ and the flow τ_2 never joins path of flow τ_1 after leaving it at node 6.

All flows are scheduled with a FIFO (First In First Out) algorithm in every visited node (non preemptive policy). Each flow τ_i has a minimum gap time between two consecutive packets at ingress node h , denoted as T_i , a maximum release jitter at the ingress node denoted as J_i (it is the duration between the packet arrival time and the time it is taken into account by the scheduler), an end-to-end deadline D_i which is the maximum end-to-end response time acceptable and a maximum processing time C_i^h on each node h , with $h \in P_i$. The transmission time of any packet on any link between nodes has known lower and upper bounds Tr_{min} and Tr_{max} (corresponding to the minimum packet size S_{min} and maximum packet size S_{max} respectively) and there are neither collisions nor packet losses on links. This is illustrated in figure 3.5.

The end-to-end response time of a packet is the sum of the times spent in each crossed nodes and the transmission delays on links. The transmission delays on links are upper bounded by Tr_{max} . Considering the FIFO scheduling, the time spent by a packet m in a node h depends on the pending packets in h at the arrival time of m in h (because all these pending packets have a higher priority than m due to FIFO scheduling and, thus, will be processed before m). The problem is then to upper bound the overall time spent in the visited nodes. The solution proposed by the Trajectory approach is based on the *busy period* concept. A busy period of level \mathcal{L} is a time interval $[t, t']$ within which jobs of priority \mathcal{L} or higher are processed throughout the period $[t, t']$ but no jobs of priority \mathcal{L} or higher are present just before and after the period $[t, t']$. In simple words, busy period can be considered as the time duration during which port

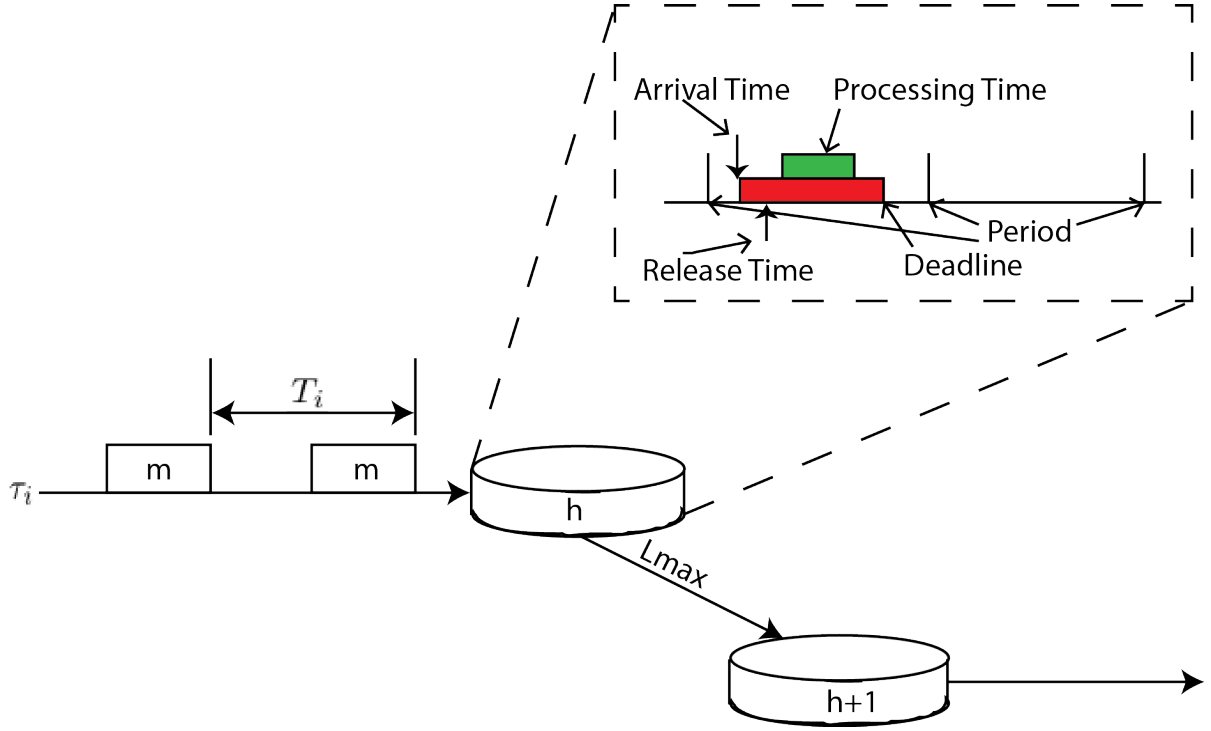


Figure 3.5 – Model used by Trajectory approach.

is busy continuously.

The Trajectory approach considers a packet m from flow τ_i generated at time t , it identifies the busy period and the packets impacting its end-to-end delay on all the nodes visited by m . It enables the computation of the latest starting time of m on its last node. This computation will be illustrated in the context of AFDX network in next section.

3.1.2.2 Application to AFDX

The Trajectory approach is applied to AFDX network in the following way:

- Each AFDX switch output port including the output link becomes a *node* of Trajectory approach.
- The switching latency of AFDX switches is represented by links of trajectory approach.
- A VL path of AFDX network corresponds to a *flow* in Trajectory approach.

Assumptions of Trajectory approach are satisfied in AFDX network:

- AFDX switch output ports implement FIFO service discipline, which satisfies the assumption of FIFO service discipline used in Trajectory approach.
- AFDX switching latency is upper bounded by a fixed value ($16 \mu s$), hence $L=L_{min}=L_{max}=16 \mu s$.
- There are no collisions on the AFDX networks due to Full Duplex links and buffers are dimensioned so that no packet is lost.
- AFDX network is configured so that a VL path never crosses another VL path more than once.
- Routing of the VLs is statically defined.
- VL parameters match the definition of flow in Trajectory approach *i.e.* $T_i = BAG$, $C_i^h = S_{max}/R$, $J_i = 0$ and $R = 100 \text{ Mb/s}$

To explain how the trajectory approach works on AFDX network, a simple example will be used to illustrate the concept.

Trajectory Approach Example: Let's consider an example of AFDX network shown in figure 3.6, in order to illustrate the Trajectory approach theory. We consider that:

- All the flows have identical characteristics : $BAG = 4000 \mu s$ and $S_{max} = 4000 \text{ bits}$.
- The entire network works at $R = 100 \text{ Mb/s}$ and the technological latency in an output port is $L = 16 \mu s$.
- There are five end systems ($e1$ to $e5$) which are sending data and two end systems which are receiving the data ($e6$ and $e7$).
- Each sending end system emits one VL. All VLs arrive at end-system $e6$, except for $v2$ which ends at end-system $e7$.

As discussed before, Trajectory approach is based on the *busy period* concept, therefore we must determine busy periods of the AFDX network shown in figure 3.6. Figure 3.7 shows an arbitrary scheduling of the packets of AFDX network in figure 3.6. The packet of a VL vi is denoted as i . Packet 3 is under study. Time of origin is chosen as the arrival time of packet 3 on node $e3$ (denoted as a_3^{e3}). After being processed in node $e3$ and after a $16 \mu s$ switching latency delay, the packet arrives at node $S2$ at time $a_3^{S2} = 56 \mu s$. Packet 4 arrives on node $S2$ at time $a_4^{S2} = 20 \mu s$ and is immediately processed. As packet 3 arrived after packet 4 it has to wait until the output port is freed by packet 3. Packet 4 arrives at node $S3$ at time $a_4^{S3} = 76 \mu s$

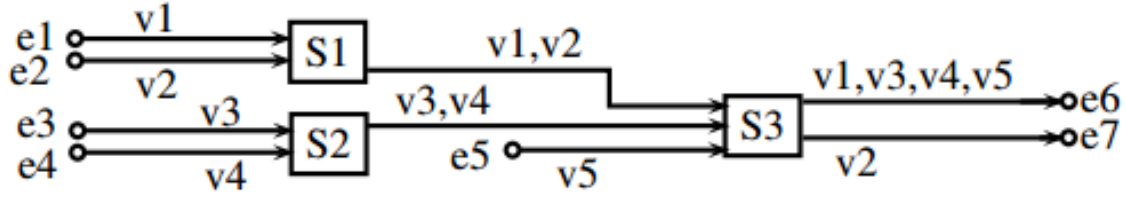


Figure 3.6 – AFDX network for Trajectory approach example.

where it is processed after packet 1 and packet 5 which arrived before it at time $a_1^{S3} = 5\mu s$ and $a_5^{S3} = 38\mu s$ respectively. Packet 3, which is the last packet to be processed in the busy period bp^{S3} arrives at node $S3$ at time $a_3^{S3} = 116\mu s$.

Packet 3 from flow $v3$ crosses three busy periods (bp^{e3} , bp^{S2} and bp^{S3}) on its trajectory. Let us consider bp^{S3} , the busy period of level corresponding to the priority of packet 3 in which packet 3 is processed on node $S3$. Let $f(S3)$ be the first packet processed in bp^{S3} with priority higher than or equal to this packet 3. As the flows may follow different paths in the network depending upon the static routing defined by AFDX network, therefore it is possible that packet $f(S3)$ do not come from the same previous node as packet 3 (which is true in this example, because packet 1 comes from node $S1$ and packet 3 comes from node $S2$). We then define $p(S2)$ as the first packet processed between $f(S3)$ and packet 3 that comes from the same node as packet 3 (here $p(S2)$ is packet 4 from node $S2$). Packet $p(S2)$ has been processed on node $S2$ in a busy period bp^{S2} of level corresponding to the priority of $p(S2)$. $f(S2)$ is then the first packet processed in bp^{S2} with a priority higher or equal to the priority of $p(S2)$. Here, we have $f(S2) = p(S2)$, which is not always the case. The same naming process is applied backwards until the ingress node of the VL is reached: the busy period bp^{e3} on node $e3$, of level corresponding to the priority of packet $p(e3)$ in which $f(e3)$ is processed.

Let a_m^h be the arrival time of packet m on node h and consider the arrival time of packet $f(e3)$ in node $e3$ as time of origin, then $a_{f(e3)}^{e3} = 0$. By adding parts of the busy periods crossed by packet 3 on its path, we can express the latest starting time of packet 3 in node $S3$. The calculation of which part of busy period to add is bit tricky to understand. This part is calculated in a node h , as the processing times of the packets between $f(h)$ and $p(h)$ minus the difference between the arrival time of $p(h-1)$ (denoted as $a_{p(h-1)}^h$) and $f(h)$ (denoted as $a_{f(h)}^h$). Hence, part of the busy period to consider $= \sum C_m^h - (a_{p(h-1)}^h - a_{f(h)}^h)$ where C_m^h is the processing time or transmission time of packet m in node h . Let us apply this to our example in 3.7.

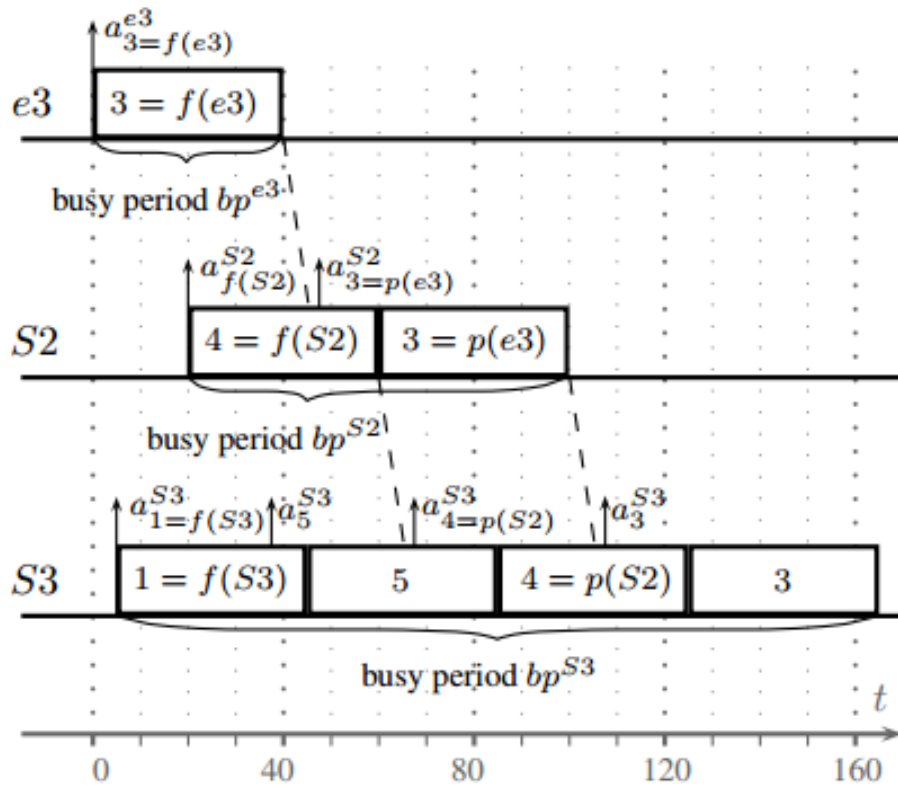


Figure 3.7 – Identification of busy periods.

- In node $e3$, $f(e3) = p(e3)$ and it is first node therefore $a_{p(e3-1)}^{e3} - a_{f(e3)}^{e3} = 0$, so we count: C_3^{e3}
- In node $S2$, $f(S2) = p(S2)$. Thus, we count: $C_4^{S2} - (a_{p(e3)}^{S2} - a_{f(S2)}^{S2})$
- In node $S3$, there is no packet $p(S3)$. Thus, we count the packets from $f(S3)$ until the packet before packet 3: $C_1^{S3} + C_5^{S3} + C_4^{S3} - (a_{p(S2)}^{S3} - a_{f(S3)}^{S3})$
- Finally, if we add the transmission times between the nodes, we can get the latest starting time of packet 3 on node $S3$, which is :

$$a_3^{S3} = C_3^{e3} + L + C_4^{S2} - (a_{p(e3)}^{S2} - a_{f(S2)}^{S2}) + L + C_1^{S3} + C_5^{S3} + C_4^{S3} - (a_{p(S2)}^{S3} - a_{f(S3)}^{S3}) \quad (3.8)$$

For our example, we have $C_h^m = C = S_{max}/R = 40\mu s$ and $L_{min} = L_{max} = L = 16\mu s$. Using these values in equation 3.8, we get:

$$\begin{aligned} a_3^{S3} &= 5C + 2L - (a_{p(e3)}^{S2} - a_{f(S2)}^{S2}) - (a_{p(S2)}^{S3} - a_{f(S3)}^{S3}) \\ &= 5 \times 40 + 2 \times 16 - (56 - 20) - (76 - 5) \\ &= 125\mu s \end{aligned} \quad (3.9)$$

For the worst case scenario we need to maximize this latest starting time of packet 3. According to the Trajectory approach presented in [Martin 2006a], we can do this by ignoring term $(a_{p(h-1)}^h - a_{f(h)}^h)$ for every node h on the considered path. This means that the arrival time of every packet coming from another preceding node should be postponed in order to increase the departure time of packet 3 in it's last node. The effect of this postponing is illustrated in figure 3.8. More precisely, the arrival time of packet 4 at node $S2$ has been postponed to the arrival time of packet 3 at node $S2$ ($a_4^{S2} = a_3^{S2} = 56\mu s$). In node $S3$, packets 1 and 5 have been postponed in order to arrive between packet 4 and 3, therefore: $a_4^{S3} \leq a_1^{S3} \leq a_5^{S3} \leq a_3^{S3}$. The worst case end-to-end delay of packet m is the sum of it's latest starting time on it's last visited node and the processing time of the packet in this last node. Thus the maximum end-to-end delay of m is: $L + C_3^{e3} + C_4^{S2} + L + (C_1^{S3} + C_5^{S3} + C_4^{S3}) + C_3^{S3} = 6C + 2L = 272\mu s$. Trajectory approach, like Network calculus, is pessimistic and can lead to impossible scenarios. Nonetheless, it does provide sure upper bounds on worst case end to end communication delays of AFDX network.

3.1.3 Pessimism of Network calculus and Trajectory approach

Lot of research has been made since the first application of network calculus to AFDX network in order to improve the results. In this regard, the results were further improved by tightening

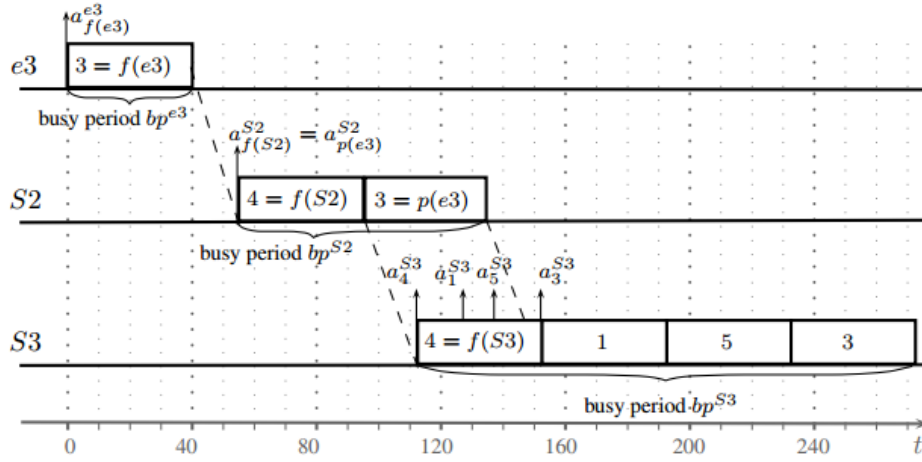


Figure 3.8 – Maximizing the arrival time in last node.

the end to end bounds by Marc Boyer and Christian Fraboul in [Boyer 2008] by using the "grouping" technique. In this technique, we "group" the VLs that exit from the same switch output port and enter another switch together, i.e. Virtual Links that share two segments of path at least. The key issue is that the frames of those VLs are serialized once exiting the first multiplexer and thus they don't have to be serialized again in the following switches. This optimization always gives tighter bounds. Another improvement in network calculus for AFDX was done by Xiaoting Li in [Li 2010] where the authors incorporated the local scheduling in the end systems into network calculus by using *offsets*. This resulted in further reduction in bounds of end to end delays calculated by network calculus.

The Trajectory approach was first applied to AFDX network in [Bauer 2009]. Later on the results were further improved in [Bauer 2010] by using the concept of "grouping", just like network calculus where VLs that share same segments of the network are grouped together. On average, Trajectory approach calculates tighter bounds as compared to network calculus (but in some cases, Network calculus has tighter bounds than Trajectory approach) and hence it improves the end-to-end delays of AFDX network calculated earlier with help of network calculus. Still, the results of Trajectory approach are pessimistic, and the measure of this pessimism was estimated in [Bauer 2010] and concluded that the trajectory approach is at least two times less pessimistic than the Network Calculus approach. Also, the upper bound of pessimism in Trajectory approach varies from 0% to 33% with average of about 7% (remember that without knowing the exact worst case delay, it is not possible to find exact pessimism as illustrated in figure 1.2).

3.1.4 Conclusion

Network calculus and Trajectory approach have relatively less computational complexity in algorithms used for calculations but both methods calculate pessimistic sure upper bounds on end to end communication delays. Presently, network calculus is being used for certification on commercial aircraft such as Airbus A380 because it was the first approach used for calculating end to end communication delay bounds. Trajectory approach, on average, gives better results as compared to network calculus. Both methods don't provide us exact end to end communication delays, therefore, we need other methods to find exact communication delays. In the next section, we will talk about the methods which give us exact end to end communication delays.

3.2 Exact Worst Case End-to-End Communication Delays

A second approach to guarantee the maximum transmission delay in AFDX network is to find the exact worst case end to end communication delays. Finding exact worst case communication delays can be classified into two main approaches: a *formal methods* approach of model checking and exhaustive simulation approach. Both approaches require some form of *modeling* in order to transform AFDX network into a model which underlying approach can work with. In simple words, in order to find exact worst case communication delays, one must check all possible cases or scenarios. These cases or scenarios are often referred to as *state-space*. Theoretically, the state-space of AFDX communication network is infinite (because of continuous real-time nature of the network) therefore these approaches also require some kind of state-space reduction. In the following sections, we will present these approaches.

3.2.1 Model Checking

Model checking can be used to find exact worst case communication delays of AFDX network. We need to model the AFDX network using model description language of the model checker being used for this purpose. Then, we need to formalize some properties using the property specification language of the tool/model checker being used. And finally, we run the model checker to check the property we have formalized by using the AFDX network model.

3.2.1.1 Application of Model checking on AFDX Network

As we have seen in section 2.3 that an AFDX network is a real-time system which requires timing constraints. So to model an AFDX network, we need to model *time*. Also we are interested in finding exact end to end communication delays, therefore we must model how the communication works in AFDX network. This includes the behavior of each end system which generates packets to be transmitted on the AFDX network. To model these entities of the AFDX network we need:

- A language to model the behavior of the system *i.e* AFDX network (including *time*, communication and end system etc).
- A language to describe the properties which we want to verify, such as what will be the maximum delay from the transmission of a packet from the source to the reception of this packet at the destination? This can be different than the modeling language of the system or it can be same.
- A tool to check that the system model satisfies the property we desire to verify.

For this purpose, two kind of model checkers can be used for AFDX network communication delay analysis: Real-time model checkers and Simply timed (or discrete time) model checkers. Further detail about these types and related software can be found in Appendix A. AFDX network can be modeled as simply timed system. Hence we can use both real-time model checkers and Plain or un-timed model checkers with Explicit time model checking approach. But AFDX network is a large real time system and its not feasible to use Plain model checkers with explicit time technique for AFDX network due to huge state space contributed by explicit modelling of time [Adnan 2010a]. This section will not describe each model checker listed in the table A.1 of Appendix A but will focus only on model checkers with potential for use in AFDX network communication delay analysis. Two prominent model checkers suit best for this job: Timed Automata based model checkers (using UPPAAL [UPPAAL] tool) from real-time model checkers category and NuSMV [NuSMV] from simply timed (discrete clock) model checkers. In the following sections we will see how to model AFDX network in these tools and how to evaluate worst case end-to-end communication delays.

3.2.1.2 NuSMV and AFDX Network

NuSMV is a symbolic model checker developed as a joint project between the Formal Methods group in the Automated Reasoning System division at ITC-IRST, the Model Checking group

at Carnegie Mellon University, the Mechanized Reasoning Group at University of Genova and the Mechanized Reasoning Group at University of Trento. NuSMV is a reimplementation and extension of SMV, the first model checker based on BDDs (Binary Decision Diagrams). NuSMV has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas. NuSMV2, combines BDD-based model checking component that exploits the CUDD library developed by Fabio Somenzi at Colorado University and SAT-based model checking component that includes an RBC-based Bounded Model Checker, connected to the SIM SAT library developed by the University of Genova.

The main features of NuSMV are:

- **Functionalities.** NuSMV allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based(Binary Decision Diagram) and SAT-based(Boolean Satisfiability) model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried on with a textual interface, as well as in batch mode.
- **Architecture.** A software architecture has been defined. The different components and functionality of NuSMV have been isolated and separated in modules. Interfaces between modules have been provided. This reduces the effort needed to modify and extend NuSMV.
- **Quality of the implementation.** NuSMV is written in ANSI C, is POSIX compliant, and has been debugged with Purify in order to detect memory leaks. Furthermore, the system code is thoroughly commented. NuSMV uses the state of the art BDD package developed at Colorado University, and provides a general interface for linking with state-of-the-art SAT solvers. This makes NuSMV very robust, portable, efficient, and easy to understand by people other than the developers.

The input language of NuSMV is designed to allow for the description of Finite State Machines (FSMs) which range from completely synchronous to completely asynchronous, and from the detailed to the abstract. One can specify a system as a synchronous Mealy machine, or as an asynchronous network of non-deterministic processes. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones: Boolean, scalars and fixed arrays. Static data types can also be constructed.

The primary purpose of the NuSMV input is to describe the transition relation of the FSM; this relation describes the valid evolution of the state of the FSM. In general, any propositional expression in the propositional calculus can be used to define the transition relation. This provides a great deal of flexibility, and at the same time a certain danger of inconsistency. For example, the presence of a logical contradiction can result in a deadlock; a state or states with no successor. This can make some specifications vacuously true, and makes the description unimplementable. While the model checking process can be used to check for deadlocks, it is best to avoid the problem when possible by using a restricted description style. The NuSMV system supports this by providing a parallel-assignment syntax. The semantics of assignment in NuSMV is similar to that of single assignment data flow language. By checking programs for multiple parallel assignments to the same variable, circular assignments, and type errors, the interpreter insures that a program using only the assignment mechanism is implementable. Consequently, this fragment of the language can be viewed as a description language, or a programming language. Comprehensive details of NuSMV input language can be found in [NuSMV].

To model AFDX network in NuSMV, we need to abstract basic and necessary characteristics of AFDX network. We should not model every minute detail of AFDX network because it will result in huge model which will not be good for model checking. On the other hand, too much of abstraction can lead to incomplete model and hence will give false results. Therefore it is very important to model the AFDX network in best possible way for model checking purposes. In the paragraphs below, the modeling approach is described for NuSMV along with the results.

Modeled Characteristics. Basic characteristics for the AFDX network, necessary for modeling purpose are:

- End Systems
 - Output port
 - Scheduling at output port
 - Packet size
- Switches
 - Input buffer
 - Output port
 - Packet queues and FIFO functionality

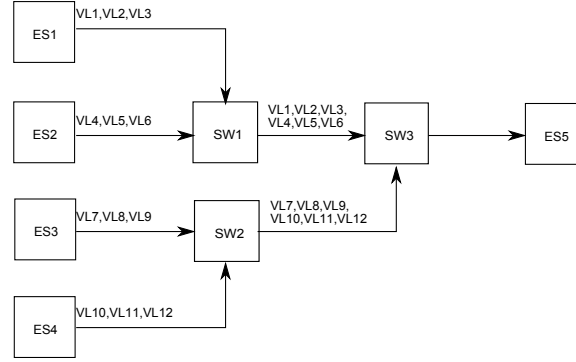


Figure 3.9 – Schematic view of sample AFDX network for NuSMV model.

Functionality of each End System is modeled with help of different NuSMV modules. For example, each VL is modeled as a separate module in NuSMV. Similarly functionality of Switch is also modeled with different modules. Each path within the switch from input to output port for each VL is modeled with a module. The process is explained with help of sample AFDX network in figure 3.9. Please note that values for VL periods and packet sizes (equivalent transmission times) are not strictly coherent with AFDX standard. They were chosen differently for purely comparative study purpose. There are five end systems and three switches. End system ES1 to ES4 transmit 3 VLs each and end system ES5 receives all of these VLs through switch SW3. AFDX network is modeled with the analogy of CPU tasks, working on basis of *request* and *granted* mechanism with dependency structure. Each end system implements a scheduling of its VLs, as illustrated in Figure 3.11a. Frames are generated in a known order with offsets between their generation times. It corresponds to the modelling of the Network Calculus approach proposed in [Li 2010] as mentioned in paragraph 3.1.1.2. So, each end system sends its first VL packet and then wait for the offset period before sending the next VL packet and so on. Offsets are modeled in NuSMV with timers and at the end of each timer, the corresponding VL task is triggered.

Each VL model in NuSMV has four parameters; *timeout*, *processor_granted*, *request* and *finish*. *timeout* is used with internal variable *state* to trigger the start of VL. *state* is also used to count the execution time. In NuSMV, each transition takes one unit time, so length of *state* represents execution time of the VL, or in this case it represents transmission time of the VL. *processor_granted* is used to indicate that output port is free so VL can start its transmission. *request* is used to model the readiness of the VL and *finish* is used to indicate when VL has transmitted its data. This is shown in code snippet in figure 3.10. For VLs, modules are named as *VLnm* where *n* represents end system number and *m* represents the VL number of this end system, e.g. *VL11* means first VL of end system *ES1*. Initial state is 0, defined by line 16 in figure 3.10. Module stays in this state till the *timeout* signal is received (this signal is coming

```

1  ---
2  --- ES 1. Period 20ms, 3 VLs;exec time: v11: 3ms, v12: 3ms, v13: 4ms.
3  ---
4  -- VL 11: period 20ms, execution time 3ms
5  MODULE VL11(timeout, processor_granted, request, finish)
6  VAR
7      state: 0..3;                -- program counter.
8  DEFINE
9      start := state = 0 & timeout;
10     finish := state = 3;
11     request := case
12         state = 0: 0;            -- Tells the scheduler that this
13         1: 1;                    -- process wants to execute.
14     esac;
15 ASSIGN
16     init(state) := 0;
17     next(state) := case
18         start: 1;
19         finish: 0;
20         processor_granted != v11: state;
21         state = 0: 0;
22         1: state + 1 mod 4;
23     esac;

```

Figure 3.10 – NuSMV code for modeling a VL.

from the timer to indicate that VL is ready to transmit according to the scheduling of VLs, as shown in figure 3.11a). As soon as *timeout* signal is received, the module starts, indicated by *start* defined on line 9 in figure 3.10 and goes to state 1 as defined on line 18. The module will continue to increase *state* by one (line 22) whenever it has the *processor_granted* signal (line 20). In the case where this VL is interrupted due to the higher priority VL, the module will wait in its current state (line 20). Since the *VL11* in code snippet in figure 3.10 has execution time of 3 time units, the *state* variable has length ranging from 0 to 3. When we have reached *state* 3, the module indicates that VL has finished its transmission, indicated by line 10 and we go back to initial state (line 19) and wait for the next period (next activation of *timeout* signal).

One timer is used for each end system to control the transmission of first packet by the end system, *e.g* for ES1 the timer *timeoutT1* is used. After the transmission of first packet of a given end system (*e.g* VL1 of ES1), the first offset timer is triggered, in above case *P11finish* triggers *offset11* timer which models the offset between VL1 and VL2. Similarly, end of *offset11* timer triggers transmission of VL2 which in turn triggers *offset12* timer (representing offset between VL2 and VL3) and so on. This process repeats for all VLs of the end system in an infinite loop. The process is shown in figure 3.11a. The packet size of each VL is modeled as execution time of the task modeled for this VL.

Similar methodology is adopted for the model of AFDX Switch with a major difference of task priorities. In models of end system, all VLs have fixed sequence and offsets between

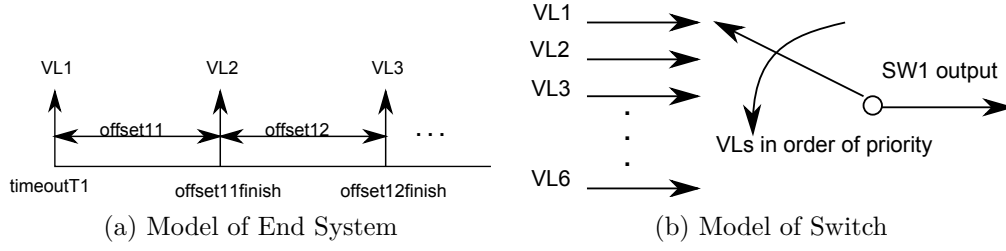


Figure 3.11 – Modeling of VLs at the End System and Switch.

them, ensuring that there will never be a simultaneous access for the output port, hence there is no need of priority and port access mechanism. But in case of switch, more than one VL from different end systems can arrive simultaneously, hence we need to model the port access mechanism so that there is no conflict and collisions. In real AFDX network this is done with FIFO buffers, so VLs are stored in order of their arrival and transmitted sequentially. To model this functionality in NuSMV, we make some assumptions to simplify the model (we can model FIFO in NuSMV but it makes model more complicated and large for verification). We assume that instead of a FIFO we have priorities and hence a VL with higher priority is transmitted first if there is a case of simultaneous arrivals. This is depicted in figure 3.11b. SW1 output port is shared by VL1 to VL6 and access is granted to any VL if the port is not busy. In case of simultaneous access, the higher priority VL gets the access first *e.g* if VL VL1 and VL3 arrive together, then VL1 will be transmitted first and VL3 later. We assume that the overall traffic load is light and hence there will not be a case where a low priority VL will always be waiting for access. This property is also verified with model checker.

Each end system has its own clock and it is not synchronized with other end systems or switches, so there is no synchronization among end systems and switches. This fact can also be modeled with NuSMV by initializing the end system timers with a set of all possible values, which in the case of AFDX network will translate into a set of integers with values from zero to the highest period in the network, theoretically it is 128 *ms*. This is shown in code snippet in figure 3.12 line 13 to 15.

Finally the whole model is checked and delay is calculated using NuSMV verification language, such as queries below find the minimum and maximum time between start of VL1 at ES1 and reception of VL1 at ES5.

```
COMPUTE MIN[SW11.start, sw3P13finish]
COMPUTE MAX[SW11.start, sw3P13finish]
```

```

1  --
2  -- Main Module
3  --
4  MODULE main
5
6  VAR
7      timer1: 0..20;          --timer for ES1, with period of 20 time units
8      timer2: 0..50;          --timer for ES2, with period of 50 time units
9      timer3: 0..100;         --timer for ES3, with period of 100 time units
10     timer4: 0..30;          --timer for ES4, with period of 30 time units
11 ASSIGN
12     init(timer1) := {0};      --timer1 is for VL under study
13     init(timer2) := {0,128}; --asynchronous start, from 0 to 128 time units
14     init(timer3) := {0,128};
15     init(timer4) := {0,128};
16     next(timer1) := timer1 + 1 mod 20; --increment timer after one cycle
17     next(timer2) := timer2 + 1 mod 50;
18     next(timer3) := timer3 + 1 mod 100;
19     next(timer4) := timer4 + 1 mod 30;
20
21 DEFINE
22     timeoutT1 := timer1 mod 20 = 0; --timeout signal, asserted at end of each period
23     timeoutT2 := timer2 mod 50 = 0;
24     timeoutT3 := timer3 mod 100 = 0;
25     timeoutT4 := timer4 mod 30 = 0;

```

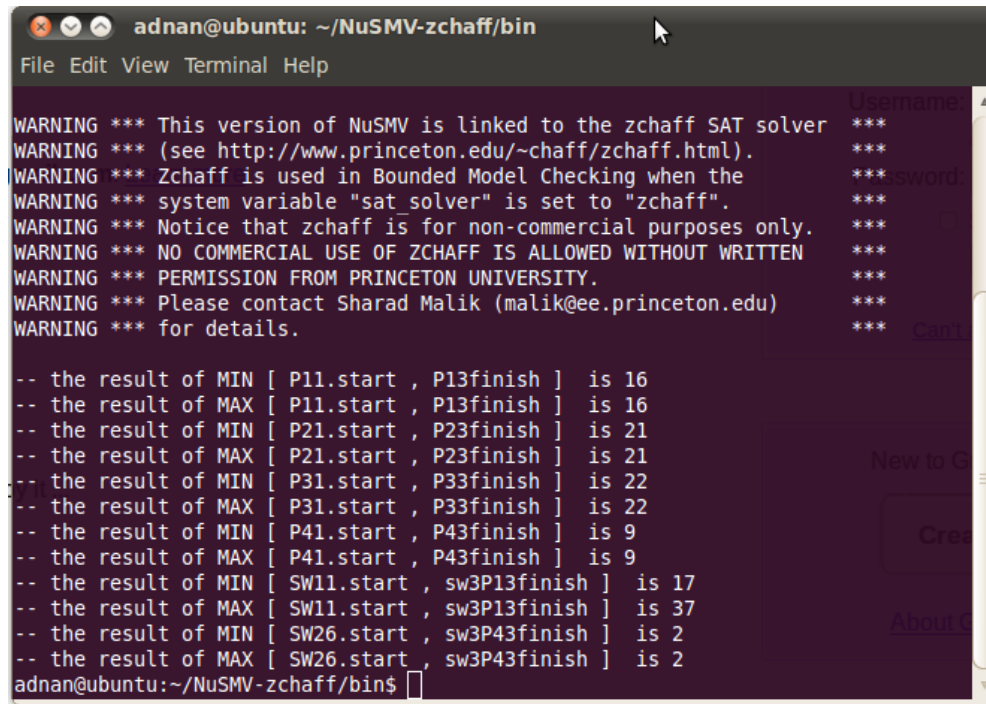
Figure 3.12 – NuSMV code for timers.

The resulting output of NuSMV model checker is shown in figure 3.13. Total running time was around two hours on 3.3 GHz Intel Core 2 Duo machine with 4 GB ram. The calculated delays are in terms of reference time step which in this case is $1ms$. NuSMV is good tool but not well suited to AFDX communication delay calculations. Main limitations are the lack of clock assignments, because NuSMV does not have clock variables. The input language of NuSMV is designed to allow for the description of Finite State Machines, and it is assumed that each transition takes unit time for execution. Hence, by counting the number of transitions and states we can measure the number of time units elapsed between two states. This also means that for modeling of large time interval or duration, number of states will increase accordingly. Also, modeling of asynchronous behavior in NuSMV has been deprecated according to latest release (version 2.5) of NuSMV ¹. According to its user manual, on page 33, it states that: “*Since NUSMV version 2.5.0 processes are deprecated. In future versions of NUSMV processes may be no longer supported, and only synchronous FSM will be supported by the input language. Modeling of asynchronous processes will have to be resolved at higher level.*”

3.2.1.3 Timed Automata and AFDX Network using UPPAAL

UPPAAL is an integrated tool environment for modeling, simulation, validation and verification of real-time systems modeled as networks of Timed Automata, extended with data types

¹<http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf>



```

adnan@ubuntu: ~/NuSMV-zchaff/bin
File Edit View Terminal Help

WARNING *** This version of NuSMV is linked to the zchaff SAT solver ***
WARNING *** (see http://www.princeton.edu/~chaff/zchaff.html). ***
WARNING *** Zchaff is used in Bounded Model Checking when the ***
WARNING *** system variable "sat_solver" is set to "zchaff". ***
WARNING *** Notice that zchaff is for non-commercial purposes only. ***
WARNING *** NO COMMERCIAL USE OF ZCHAFF IS ALLOWED WITHOUT WRITTEN ***
WARNING *** PERMISSION FROM PRINCETON UNIVERSITY. ***
WARNING *** Please contact Sharad Malik (malik@ee.princeton.edu) ***
WARNING *** for details. ***

-- the result of MIN [ P11.start , P13finish ] is 16
-- the result of MAX [ P11.start , P13finish ] is 16
-- the result of MIN [ P21.start , P23finish ] is 21
-- the result of MAX [ P21.start , P23finish ] is 21
-- the result of MIN [ P31.start , P33finish ] is 22
-- the result of MAX [ P31.start , P33finish ] is 22
-- the result of MIN [ P41.start , P43finish ] is 9
-- the result of MAX [ P41.start , P43finish ] is 9
-- the result of MIN [ SW11.start , sw3P13finish ] is 17
-- the result of MAX [ SW11.start , sw3P13finish ] is 37
-- the result of MIN [ SW26.start , sw3P43finish ] is 2
-- the result of MAX [ SW26.start , sw3P43finish ] is 2
adnan@ubuntu:~/NuSMV-zchaff/bin$

```

Figure 3.13 – Output of NuSMV model checker.

(bounded integers, arrays, etc.). The tool is developed in collaboration between the Department of Information Technology at Uppsala University, Sweden and the Department of Computer Science at Aalborg University in Denmark. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols in particular, those where timing aspects are critical.

UPPAAL consists of three main parts; a description language, a simulator and a model-checker.

- The description language is a non-deterministic guarded command language with data types (e.g. bounded integers, arrays, etc.). It serves as a modeling or design language to describe system behavior as networks of automata extended with clock and data variables.
- The simulator is a validation tool which enables examination of possible dynamic executions of a system during early design (or modeling) stages and thus provides an inexpensive mean of fault detection prior to verification by the model-checker which covers the exhaustive dynamic behavior of the system.
- The model-checker can check invariant and reachability properties by exploring the state-

space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints.

The two main design criteria for UPPAAL have been efficiency and ease of use. The application of on-the-fly searching technique has been crucial to the efficiency of the UPPAAL model-checker. Another important key to efficiency is the application of a symbolic technique that reduces verification problems to that of efficient manipulation and solving of constraints. To facilitate modeling and debugging, the UPPAAL model-checker may automatically generate a diagnostic trace that explains why a property is (or is not) satisfied by a system description. The diagnostic traces generated by the model-checker can be loaded automatically to the simulator, which may be used for visualization and investigation of the trace.

Since its first release in 1995, UPPAAL has been applied in a number of case studies (refer [UPPAAL] for more details). To meet requirements arising from the case studies, the tool has been extended with various features. The current version of UPPAAL, called Uppaal2k, was first released in September 1999. It is a client/server application implemented in Java and C++, and is currently available for Linux, SunOS, Mac OS X and Windows. The features of Uppaal2k include:

- A graphical system editor allowing graphical descriptions of systems.
- A graphical simulator which provides graphical visualization and recording of the possible dynamic behaviors of a system description, i.e. sequences of symbolic states of the system. It may also be used to visualize traces generated by the model-checker. Since version 3.4 the simulator can visualize a trace as a message sequence chart (MSC).
- A requirement specification editor that also constitutes a graphical user interface to the verifier of Uppaal2k.
- A model-checker for automatic verification of safety and bounded-liveness properties by reachability analysis of the symbolic state-space. Since version 3.2 it can also check liveness properties.
- Generation of diagnostic traces in case verification of a particular real-time system fails. The diagnostic traces may be automatically loaded and graphically visualized using the simulator. Since version 3.4 it is possible to specify that the generated trace should be the shortest or the fastest.

3.2.1.4 Modeling of AFDX Network in Timed Automata

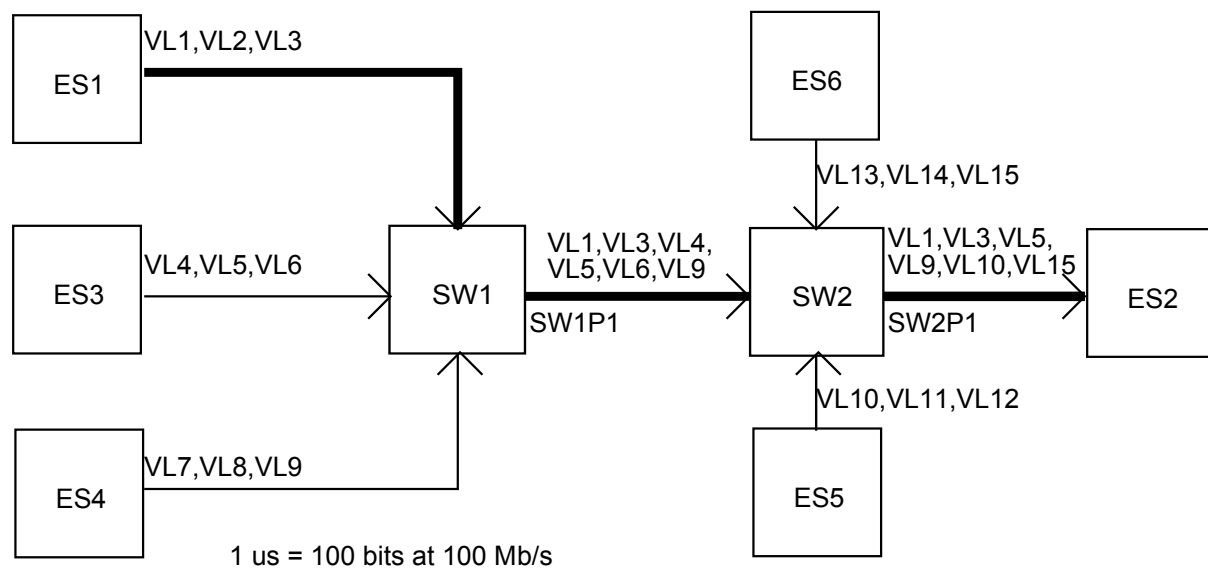
A very first effort in finding worst case communication delays for AFDX network with model checking using Timed Automata was made by Charara et al. in [Charara 2006b]. This is a first, proof of concept effort to find exact worst case delays by using timed-automate based model checking. The approach assumes a set of purely periodic VLs generated by different end systems. In order to reduce the complexity, the model was very simple and basic in its detail with respect to a real AFDX network. Each end system was assumed to have one VL and all VLs with same packet size. All traffic was assumed to be strictly periodic (real AFDX traffic is sporadic). With these simplifications the model was only able to find exact worst case delay for a very small size network with upto 5 VLs, far from the actual number of VLs in an industrial configuration of AFDX network with 1000 VLs. Nonetheless, it successfully demonstrated that model checking can be applied to AFDX network for worst case communication delays.

A model checking approach determines exact worst case delays but leads to combinatorial explosion thus restricting its use to only small networks [Charara 2006b]. To overcome this issue, a “divide and conquer” methodology was used during the work done in Master’s thesis [Adnan 2010a]. The idea is to calculate the exact worst-case delay on each output port on the path of given VL and then to propagate this delay in consecutive discrete steps starting from source node till the destination node. This is different than earlier work in [Charara 2006b] due to its port by port analysis approach. This work was published in [Adnan 2010b].

The approach is illustrated by an example of AFDX network shown in figure 3.14. This network consists of 6 End Systems interconnected with two switches. We assume that all 15 VLs of this configuration are strictly periodic. Their maximum packet size and periods are given in figure 3.14. The scheduling implemented by each ES is modeled by offsets associated to the VLs, given in figure 3.14. We focus on VL1 whose path is {ES1-SW1-SW2-ES2} (bold line in figure 3.14). There are other VLs originating from ‘ES1’ and other end systems which pass through port ‘SW1P1’ and ‘SW2P1’ (indicated by ‘x’ in columns “SW1P1” and “SW2P1”). The calculation of exact upper bound on end-to-end delay of VL1 is processed on a port by port manner; this delay is first computed at port “SW1P1” and the obtained value is then used to calculate the delay in next port “SW2P1”.

The description of the approach proceeds in 4 steps:

- Modeling of VLs and their scheduling at one end system.
- Modeling asynchronous behavior among all end systems.



VL	Size(us)	Period(ms)	Offset(ms)	SW1P1	SW2P1
VL1	30	32	4	x	x
VL2	20	32	8	-	-
VL3	30	16	0	x	x
VL4	50	32	0	x	-
VL5	20	64	8	x	x
VL6	30	128	24	x	-
VL7	20	8	2	-	-
VL8	10	4	0	-	-
VL9	30	16	6	x	x
VL10	50	64	24	-	x
VL11	40	32	8	-	-
VL12	30	16	0	-	-
VL13	20	8	0	-	-
VL14	30	16	4	-	-
VL15	30	16	12	-	x

Figure 3.14 – Example of AFDX Network configuration.

- Modeling and computation of delay at first output port “SW1P1” and
- Modeling and computation of delay at consequent output ports “SW2P1”.

Modeling of one ES behavior. As mentioned before, all VLs are assumed strictly periodic with period equal to BAG . Hence their scheduling by given ES is modeled by offsets. One VL is arbitrarily chosen as the first VL to generate a packet. Without loss of generality, we chose a VL with shortest period. In the example in figure 3.14, VL3 is the first (reference) VL for “ES1” and offsets of VL1 and VL2 are computed based on VL3 as shown in figure 3.15. Similarly VL4, VL8, VL12 and VL13 are chosen as reference VLs for end systems “ES2”, “ES3”, “ES4” and “ES5” respectively. This leads to one timed automaton for each VL. While modeling the automata, all time values are scaled by $10 \mu s$ in order to reduce time required by UPPAAL tool for verification. During verification, model checker will check models at each value of the time in a given interval, and scaling the values of time will reduce the search space.

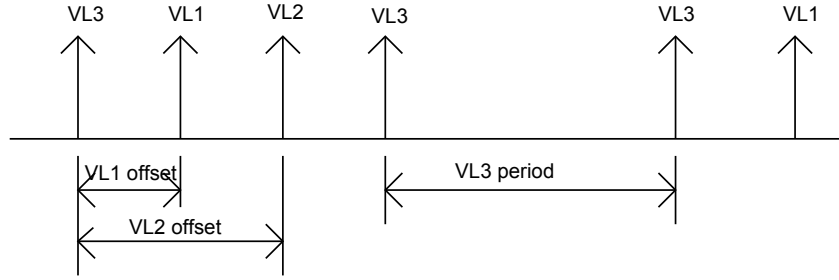


Figure 3.15 – Offsets between VLs of one ES.

For each end system, we have a set of VLs originating from this end system. We have one automata for each VL. For the first VL of each end system, we start the VL automata as soon as that end system is ready for transmission. The “end system ready to transmit” trigger is modeled by incrementing an integer variable *e.g* for “ES1” the variable is “scramble1” and it is incremented by the transition as shown in figure 3.16a. For the model of first VL of the end system, *e.g* for VL3 of ES1, the automata is depicted in figure 3.16b. After the trigger of ES1, indicated by “scramble1 > 0”, we wait for the offset time to elapse (which in case of reference VL is zero) and then we go to *jitter* state which models jitter in the VL (for VLs of originating ES, this jitter is zero *i.e* there is no jitter for VLs at the end system). The transition out of *jitter* state is non deterministic and can be taken at any time from 0 till j . This means that model checker will verify all values of jitter from 0 till j . Finally, we store the packet in FIFO queue by using function *enqueue(id)* by passing VL id as parameter (for VL3, id=3), and wait till the VL period is elapsed. After this period has elapsed, we come back to the *jitter* state and this process is repeated periodically. Clock variable “x” in all TAs is a local variable.

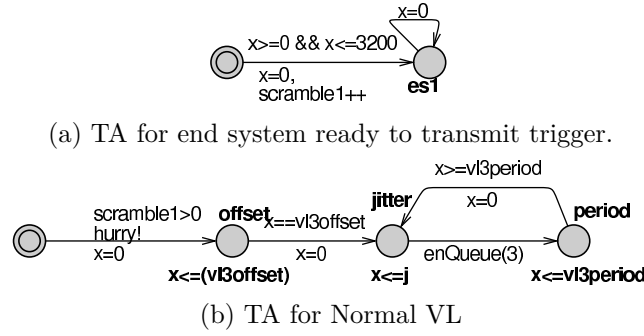


Figure 3.16 – Timed Automata for an end system.

Modeling of a set of ESs. The end systems are asynchronous with respect to each other. They can start transmitting VLs at any time with reference to other end systems, as illustrated in illustration 3.17 where “StartES1” and “StartES3” can take any value between zero and largest period of the VLs emitted by ES1 and ES3 respectively. This behavior is modeled in timed automaton in figure 3.16a representing ES1. The state “es1” is reached after a delay that can take any value between zero and 32ms (largest period of any VL of ES1 scaled by 10 μ s). The model checking will test all possible values in this range.

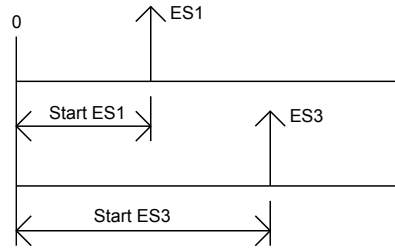


Figure 3.17 – Asynchronous behavior of ESs.

Modeling and Computation of delay at first output port. In this approach, model of first output port of a switch in the path of VL under study is different than the rest of the output ports in the path. After modeling the end systems, the next step is the computation of worst case delay at first output port *i.e* “SW1P1”. The output port is a FIFO queue (which serves packets in order of their arrival). Figure 3.18 shows timed automaton of an output port. It models the packet size, and the order among VLs. For simplicity, we assume there is no latency within the switch but it can be easily added to the model by adding a constant whose value is equal to latency of the switch. After initialization we wait in *empty* state till we receive a packet. As soon as a packet is stored in queue, we check the ID of this packet representing the VL id by using function *headQueue()* (shown in figure 3.19.) and go to corresponding state

without elapsing any time thanks to urgent channel variable “hurry”. The automaton has one location for each VL and stays in this location for the time equal to VL packet transmission time and then returns back to *empty* state removing the packet from the queue. The FIFO queue is modeled as an array with maximum size equal to upper buffer bounds calculated by network calculus. This ensures there will not be overflow in the queue. Functions *enQueue(id)* and *deQueue()* add and remove VL packets from the queue respectively. These functions are shown in figure 3.19. A value of -1 in queue indicates an empty place in the queue.

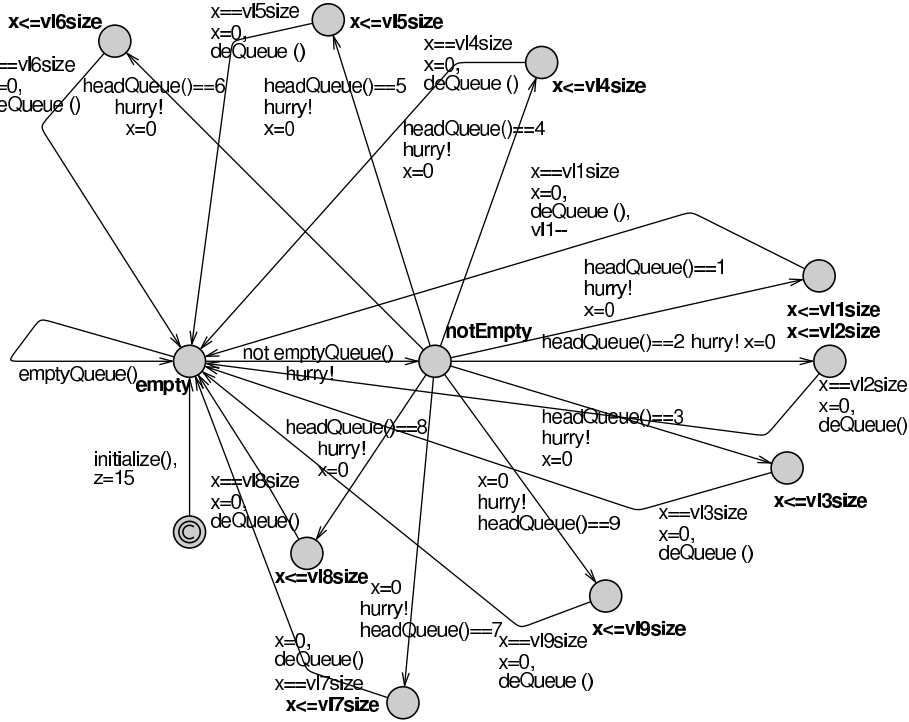


Figure 3.18 – Timed Automata for switch output port.

```

void deQueue ()
{
    for (i : int[1,queue_size-1])
        queue[i-1] = queue[i];
    queue[queue_size-1] = -1;
}

int[-1,N-1] headQueue ( )
{
    return queue[0];
}

void enQueue (int p)
{
    int i = 0;
    while (queue[i]>=0){i++;};
    queue[i]=p;
}

```

Figure 3.19 – Functions used for FIFO queue.

Modeling of VL under study. The VL under analysis requires some additional states in its model for measuring end-to-end delay. Figure 3.20 shows timed automaton of measuring VL. This automaton is very similar to the automaton of serialized VL with exception of three

additional states which are used to measure the time elapsed from transmission of packet under study at source end system till the packet is received at destination end system. The delay experienced by the packet of VL under study (in this example its VL1) is measured by local clock variable y . This clock is reset as soon as the packet has just been transmitted. The automaton models two things in the same model, the measuring of elapsed time and normal periodic transmission of VL packets. After the transmission of first packet, the automaton waits in state “fi” till either packet under study has been received at destination or it is time for the transmission of next periodic packet of this VL. The worst case delay is the smallest value x such that the value of clock y is always less than or equal to x while in the state “fi”. It is obtained by querying the timed automata using following CTL formula:

$$A[] (VL1m.fi \text{ imply } VL1m.y \leq x)$$

The value x is initialized to the sure upper bound computed by the Network Calculus and then decreased as long as the formula is verified. The delay for VL1 in “SW1P1” is $110\mu s$. We use one automaton for the two functions in order to reduce state-space. In the approach in [Charara 2006b], authors used a separate automaton for the measurements. We tested both approaches *i.e* a combined automaton for both functions, and two separate automata for each function. We found that combined automaton is efficient and has lesser state-space.

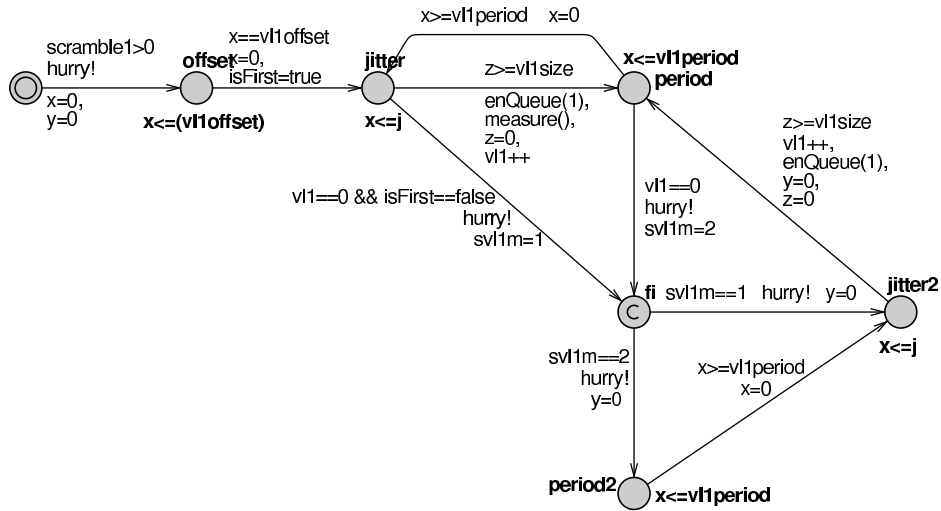


Figure 3.20 – Timed Automata for Measuring VL.

Modeling and Computation of delay at consequent ports. The waiting time in switch output port is variable and depends on the traffic at the output port. Therefore, at the output port of the switch, the sequence of the packets can be considered to have some jitter as shown

in figure 3.21a. The jitter experienced by a VL at a given point in its path is clearly the maximum waiting time of a packet of this VL in the buffers of the output ports it has crossed as illustrated in figure 3.21b. So the calculated delay at first port “SW1P1” is used as jitter j in timed automata models in second output port and so on. The above process is repeated for all output ports in the path of a given VL and whole network can be analyzed this way in port by port manner, albeit with a small addition to VL automata explained below.

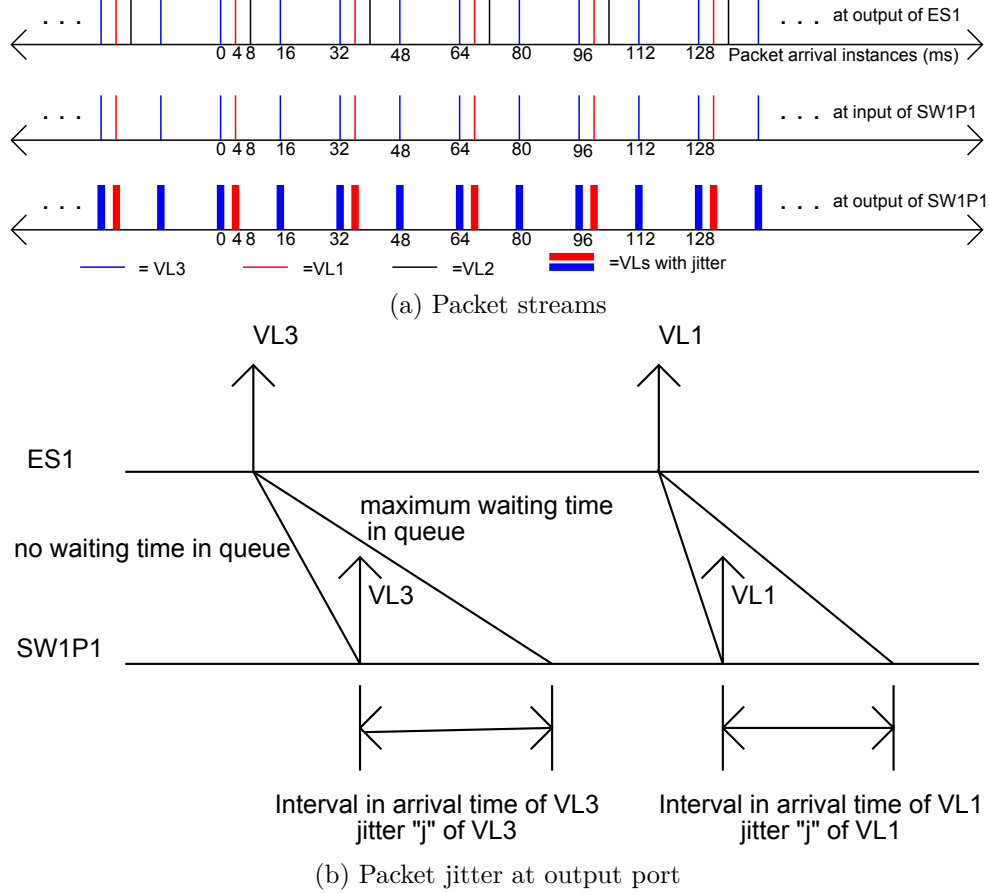


Figure 3.21 – Packet arrival instances and how a delay at output port appears as jitter.

Indeed packets sharing a link are serialized, i.e they cannot be transmitted concurrently, as illustrated in figure 3.22, where the packet of VL4 has to wait until the end of the transmission of the packet of VL1. This serialization is modeled by the clock variable z in the timed automaton in figure 3.23. It ensures that the delay between the receptions of two consecutive packets ‘P1’ and ‘P2’ is at least the transmission time for ‘P2’. This model is used for the example considered in this paper at “SW2P1” port for VL1, VL3, VL5 and VL9. The delay calculated at this port is $110 \mu s$. The end to end delay is addition of all the delays on the path of VL. In this example total delay of VL1 is transmission time at ES1+delay at SW1P1+delay at SW2P1 which equals

250 μ s. This worst case scenario is shown in figure 3.24 and it is obvious from the figure that worst case scenario occurs when VLs arrive at the same time at the switch output port *i.e* VL1 arrives at same time as VL4 and VL9 at SW1 and VL10 and VL15 arrive at the same instance as VL1 at switch SW2.

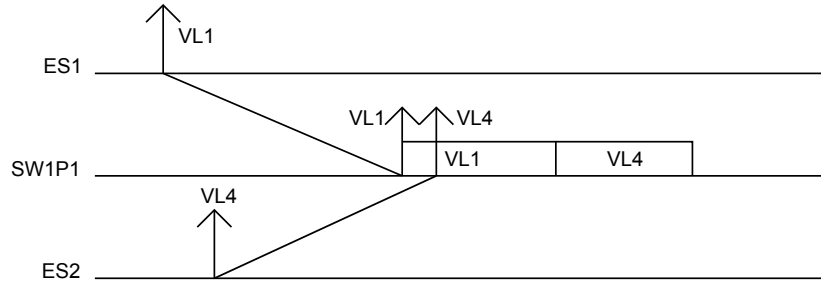


Figure 3.22 – Serialization of VLs after passing through an output port.

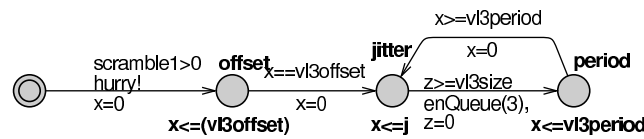


Figure 3.23 – Timed Automata for Serialized VL.

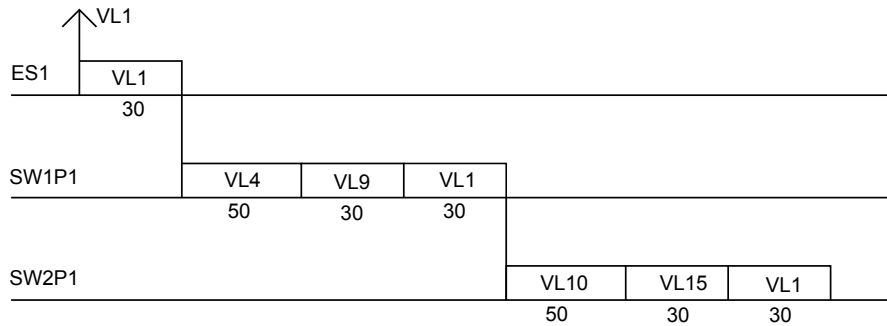


Figure 3.24 – Worst case scenario for VL1.

Results. The table 3.1 represents the results of example network. All calculations were done on a machine with 3.3 GHz Intel Core 2 Duo processor having 4 GB RAM. In general for VLs with no jitter, the computation time is about 5 seconds using approximately 15 MB of RAM but for VLs with jitters, it can take up to 15 hours using approximately 4 GB of RAM because of increased state-space due to jitters.

VL	Delay(us) at SW1P1	Delay(us) at SW2P1
VL1	110	110
VL3	110	110
VL4	110	-
VL5	80	100
VL6	90	-
VL9	110	110
VL10	-	110
VL15	-	110

Table 3.1 – Exact worst-case delays.

This “divide and conquer” or “port by port” approach seems promising in state-space reduction and model checking larger networks but this approach does not lead to the exact worst-case delay for any configuration, it can be optimistic in some scenarios because worst case delay on each port does not always lead to over all end-to-end worst case delay as demonstrated in [Adnan 2011b]. In certain cases, worst case delay at one port does not lead to worst case delay in the consequent port and sum of all delays is not the worst case end to end delay. Such a case is demonstrated in detail in Chapter 5 in Section 5.3.6. Consequently, this approach gives a valuable under estimation of the worst-case end-to-end delay, but it is not enough for certification. To over come this problem, one idea was to start modeling from destination end system and move backward to the source end system. But we found it very difficult to model this approach. It was hard to know relative start time of each end system with respect to others.

3.2.2 Exhaustive Simulation

The simulation approach has been applied to AFDX network before in [Charara 2006b] and [Scharbarg 2009] but not exhaustively in the context of exact worst case communication delays. Exhaustive simulation is very similar to model checking approach in the fact that both approaches analyze all possible cases or scenarios. Model checking uses formal methods to do this while exhaustive simulation is brute force approach which can be applied to any problem with relative simplicity and ease as compared to model checking approach. This thesis is the first effort in this direction and no prior work exist for exhaustive simulation of AFDX network for worst case communication delays. This work will be presented in detail in chapter 5.

3.2.3 Conclusion

Model checking and exhaustive simulation provide us exact end to end communication delays but the computational complexity is very high as compared to network calculus and Trajectory

approach. NuSMV is a discrete time model checker while UPPAAL is a real-time model checker. NuSMV uses symbolic representation of its state-space which is more efficient than the DBM structure used for *regions* and *zones* of timed automata state-space but we need to model time explicitly in NuSMV which results complex models and huge state-space which offsets the benefits of efficient symbolic representation of state-space. Therefore, in order to find exact end to end communication delays, timed automata approach is better than NuSMV based approach but we must improve models in order to analyze larger networks.

3.3 Conclusion

In this chapter we have seen different methods for computing end to end communication delays. Analytical methods such as Network Calculus and Trajectory approach are good in terms of resource usage (computation time and memory) but they only give us a sure upper bound of delays; these methods are pessimistic and do not provide us exact end to end communication delays. Thus efforts are underway to reduce or even eliminate this pessimism. Tighter upper bounds have been obtained by Trajectory approach [Bauer 2010]. On the other hand, model checking requires lot of computation time and resources but they can provide us with exact end to end communication delays only for very small networks with certain constraints (e.g. strictly periodic VLs only, optimistic in certain cases, buffer depth of switch output port etc). Existing model checking approaches cannot cope with industrial size configurations due to the combinatorial explosion problem.

Our aim is to find exact communication delays, therefore we can not use Network Calculus or Trajectory approach. Then, we are left with model checking approach. The main problem with approaches presented in this chapter is the huge number of possible cases, *i.e* state-space. The state-space increases exponentially with the number of VLs and detail of the model. Therefore key consideration in these approaches is to reduce the state-space and the complexity of the model of AFDX network. A very detailed model will not be able to find worst case communication delays for a large network and a very simple model will not be realistic with respect to the actual AFDX network properties. A bare minimum level of abstraction must be made in order to have a model which will satisfy the real life AFDX network properties and still be able to find worst case communication delays for larger networks. This abstraction will be presented in Chapter 4 in order to improve the worst case end to end delay computations using the timed automata theory.

An Improved Method to Compute the Exact Worst Case End-to-End Delay using Timed Automata

Contents

4.1	Characteristics of a worst-case scenario	62
4.1.1	Definition of a scenario	62
4.1.2	Critical Instance Property	62
4.2	The modelling based on timed automata	65
4.2.1	Modelling the VLs	66
4.2.2	Modelling the Switches	72
4.2.3	Modelling the Synchronization	74
4.2.4	Utility Automata: modelling of the buffers	76
4.2.5	Utility Automata: end to end delay computation	77
4.3	Limits of the approach	79
4.4	Conclusion	81

Chapter 3 shows that model checking with timed automata can be used to find exact worst case end-to-end communication delays in AFDX network. The models used were not always able to find exact worst case end to end communication delays and were optimistic in certain cases. Moreover, these models were not efficient enough to analyze large AFDX networks. In this chapter we will present new models and introduce new techniques to reduce search space in order to analyze larger AFDX networks than previous approaches. The method exploits properties of the AFDX network in order to limit the number of cases that we must check for worst case scenario. For this purpose, in the next section some properties of the AFDX network are established which will allow in reducing the number of cases that can be candidate for the worst case scenario.

4.1 Characteristics of a worst-case scenario

An AFDX network is a set of end systems interconnected by a set of switches. The delay of a frame transmitted over such a network includes the transmission times on links, the switching delays and the waiting times in output buffers. Transmission times and switching delays are constant values for a given network configuration therefore the worst-case delay occurs when the overall waiting time of the frame in the output buffers is maximized. Considering this assumption, the goal is to construct only the possible scenarios leading to the worst case end to end delays.

4.1.1 Definition of a scenario

For a given network architecture, a scenario is defined by the sequences of frames generated by the different end systems and by the instant when each end system generates its first frame. Since there is no global clock in an AFDX network, no assumption can be done concerning the generation instant of the first frame of each end system. Obviously, these generation instants have an impact on the arrival time of each frame in all the output ports that it crosses, then on the waiting time of each frame in these output ports. Thus, every possible combination of these generation instants should be considered in order to determine a worst-case scenario for a given frame. This exhaustive search has been implemented in previous timed automata based modelling [Charara 2006a]. It leads to combinatorial explosion, even for small configurations. In order to reduce the search-space, we exploit some properties of AFDX network as discussed in next section where we show that only a small subset of these combinations of generation instants is candidate for a worst-case scenario.

4.1.2 Critical Instance Property

Consider an AFDX network as shown in figure 4.1. The upper part in Figure 4.1 depicts the network architecture. The VLs are not represented in the figure, except vx which is under study. We will prove in paragraph 3.3 that a worst-case scenario for a frame of vx necessarily has the characteristics of the scenario depicted in the lower part in Figure 4.1. The frame of vx arrives at the output port of $S1$ at the same instant t_0 as a frame a coming from link $e2 - S1$ and a is arbitrarily transmitted before the frame of vx . Moreover, both frames have to wait till all the frames which have arrived at the output port of $S1$ before t_0 are transmitted. Similar situations occur at the output port of $S2$ with frames b and c and at the output port of $S3$

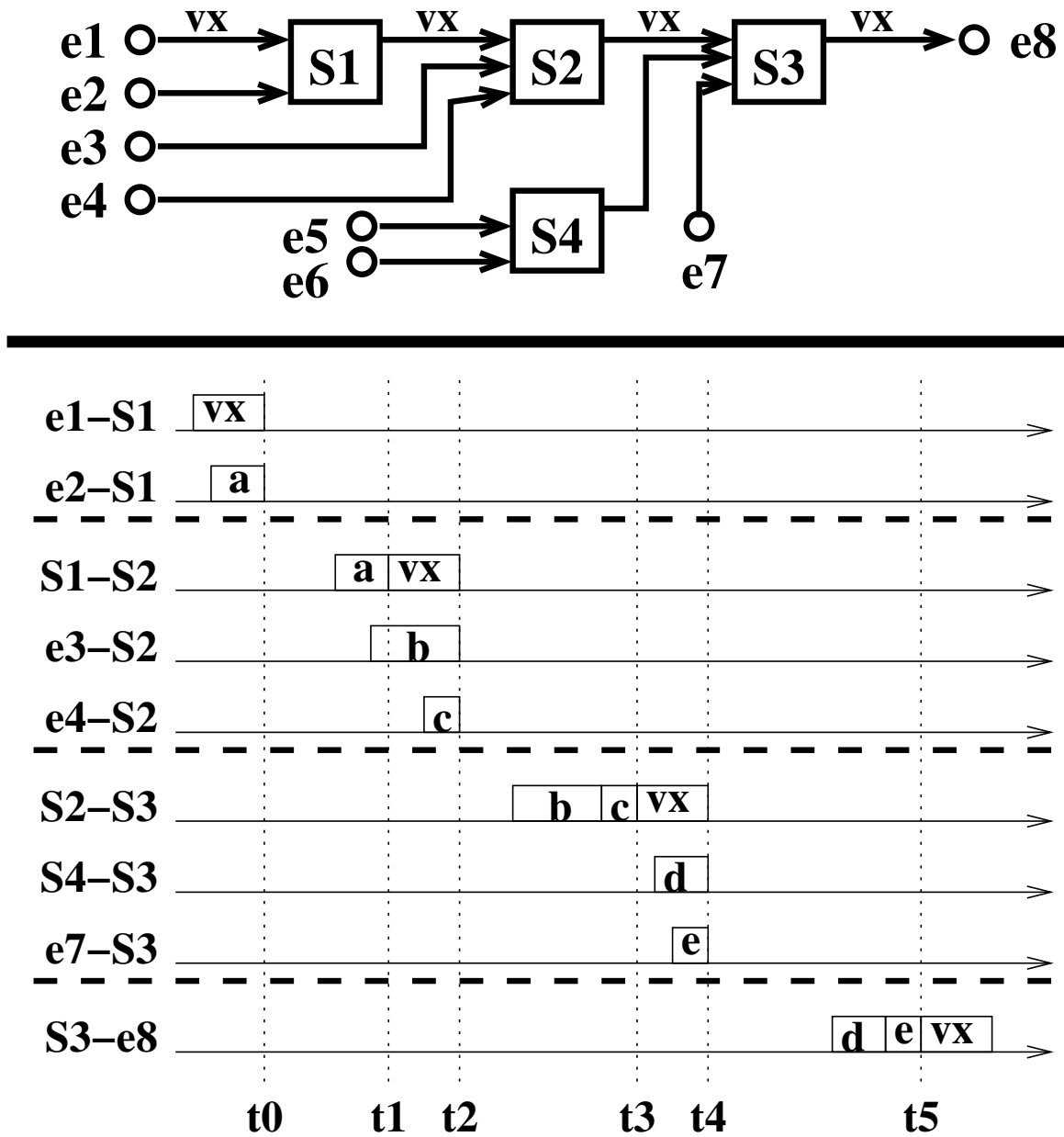


Figure 4.1 – Illustration of a worst-case scenario

with frames d and e . In the scenario of figure 4.1, vx experiences a maximum waiting delay in $S1$ since a is sent before vx . In the same way, vx is sent after b and c in $S2$ and waits for the transmission of b and c . Consequently, vx experiences a maximum waiting time in buffers if at each output port vx arrives at the same instant as one frame from all the input links of the corresponding switch and vx is transmitted at the end. this can be generalized and is known as *Critical Instance* property. Formally, we can prove this property as below:

Property 1: Critical Instance Property

The worst-case waiting time for a frame x in the output port of a switch S_i ($1 \leq i \leq n$) occurs when x arrives at the output port at the same time as one frame from each of the other input links $link(i, 2), \dots, link(i, m_i)$ and x is transmitted at the end.

Proof: To prove this property, let's consider the scenario in figure 4.2. Let's suppose that frame x from $link(i, 1)$ is the frame for which we want to calculate worst case waiting time in the switch output port of the switch S_i . If frame x does not arrive at switch S_i at the same time as one frame from *all* the other input links $link(i, j)$ ($2 \leq i \leq n$) ($2 \leq j \leq m_i$) where n represents the number of switches and m_i represents the total number of input links of a given switch S_i , then it means that for at least one input link there is no frame arriving at the output port at the same time as frame x . This is the case for frame y of $link(i, k)$ in figure 4.2.

The impact of the frames coming from $link(i, k)$ on the waiting time for x in the output port of S_i is the amount of data which has arrived at the output port up to t_0 from $link(i, k)$ and has not been transmitted at t_0 . Let's note y the last frame which is received from $link(i, k)$ before t_0 . Shifting all the frames coming from $link(i, k)$ to the right so that y arrives at the output port at t_0 does not increase the amount of data from $link(i, k)$ which are being transmitted before t_0 , since frames from $link(i, k)$ arrive later at the output port and consequently cannot be transmitted earlier. Then shifting frames from $link(i, k)$ till y arrives at the output port at t_0 can never decrease the waiting time of x at the output port. ■

Critical Instance property is an important property which allows us to reduce search space by great extent. This property will be the basis for our timed automata models which we will discuss in the coming sections.

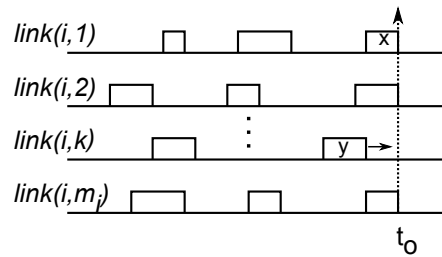


Figure 4.2 – Worst-case for a frame x.

4.2 The modelling based on timed automata

Now we propose to model the AFDX network into timed automata considering the *Critical Instance* property in order to reduce the search space of the worst case end to end delay computations. The goal of our system of the AFDX network is to compute only the scenarios which are candidate for the worst case end to end delays. Our model is composed of timed automata for:

- End systems which generate a set of VLs
- Switches which use FIFO queues to route packets

In the modelled system we focus only on a VL under study, so all the network architecture does not need to be considered: only the part which is related to the VL under study is modelled. In particular, we consider only the VLs which influence the VL under study *i.e.* VLs which share the path and output ports of the switch with the VL under study. The modelling will be composed of timed automata which models the behavior of:

- Generation of VLs. VLs can be strictly periodic and/or sporadic.
- Output ports of the switches.
- Synchronization between VLs of the network.
- *Utility* automata for measurement of end to end delay, global variables and other glue logic that may be necessary

This modelling is based on timed automata using UPPAAL software version 4.1.7, freely available from <http://www.uppaal.org/>.

While modelling the AFDX network, following assumptions are considered:

- Each end system generates a set of VLs.
- Periodic VLs are strictly periodic.
- Possible periods can only be one of the value from 1, 2, 4, 8, 16, 32, 64, 128 *ms* (AFDX network allows only these values as BAG).
- Scheduling of frames at each end system is known for periodic VLs.
- Multicast VL is treated as equivalent of individual unicast VLs corresponding to each path of the multicast VL.
- Since there is no loop in an AFDX network, the architecture corresponding to a given VL includes exactly one output link per switch.

In an AFDX network, we have mix of both strictly periodic and sporadic VLs. Some of the VLs generated by a given end system are strictly periodic. The scheduling of these VLs by the end systems is known and it can be dealt with offsets. The rest of the VLs are sporadic and their frames can be generated at any time, provided that they respect the minimum gap between two consecutive packets given by the *BAG* value.

As an example the network architecture of Figure 4.3 will be used for modelling. It consists of 5 switches (S1, S2, S3, S4, S5), 12 end systems (e1 to e12) and 32 VLs (from *v1* to *v32*). The configuration of the VLs is shown in table 4.1. VL *v19* and *v20* are sporadic while rest of the VLs are strictly periodic with offsets.

Figure 4.4 shows a possible sequence of VLs generated by the end system *e1*. This sequence is periodic. On the other hand, *e5* generates one periodic VL (*v15*) and 2 sporadic VLs (*v19* and *v20*). Thus, for these VLs, no temporal relationship can be made between the generation time of the frames.

Every part of the network in Figure 4.3 (VL generation, switch output ports, synchronization) will be modelled into timed automata in the following sections. The VL under study is *v1*, for which we will compute the worst case end to end communication delay.

4.2.1 Modelling the VLs

Automata of the VLs fall into different groups as far as modelling is concerned:

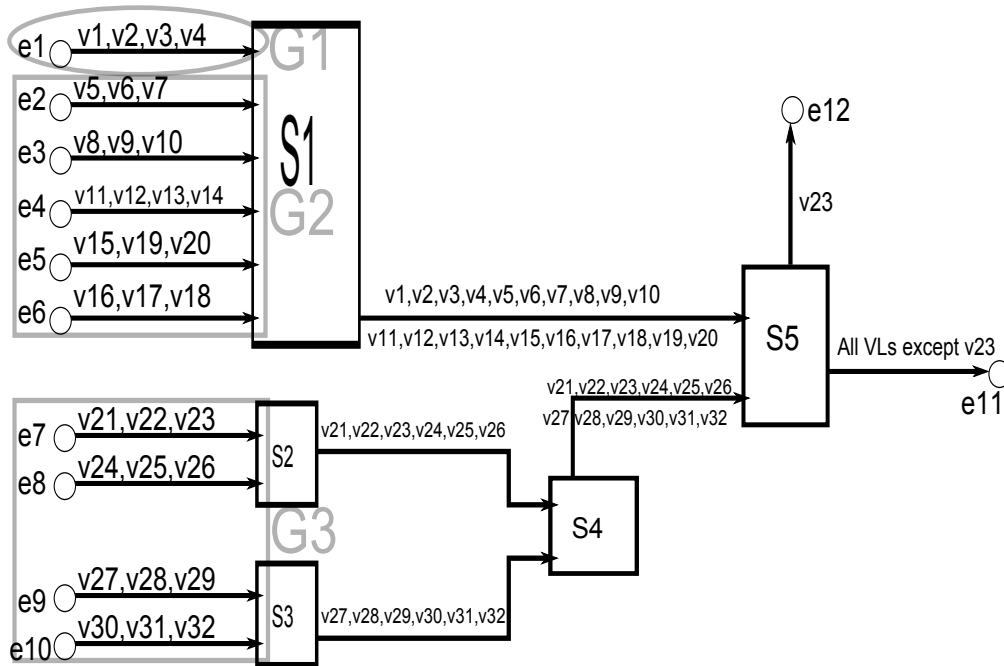


Figure 4.3 – AFDX Network architecture for improved timed automata.

VL	BAG (ms)	Size (ms)	offset (ms)
v1	128	2	0
v2	128	2	32
v3	128	2	64
v4	128	2	96
v5	128	2	0
v6	128	2	64
v7	128	2	96
v8	128	2	0
v9	128	2	32
v10	128	2	96
v11	128	2	0
v12	128	2	32
v13	128	2	64
v14	128	2	96
v15	128	2	0
v16	128	2	0

VL	BAG (ms)	Size (ms)	offset (ms)
v17	128	2	32
v18	128	2	64
v19	32	2	n/a
v20	64	2	n/a
v21	128	2	0
v22	128	2	32
v23	128	2	64
v24	128	2	0
v25	128	2	64
v26	128	2	96
v27	128	2	0
v28	128	2	64
v29	128	2	96
v30	128	2	0
v31	128	2	64
v32	128	2	96

Table 4.1 – AFDX network configuration data for improved timed automata

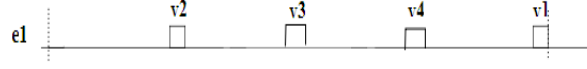
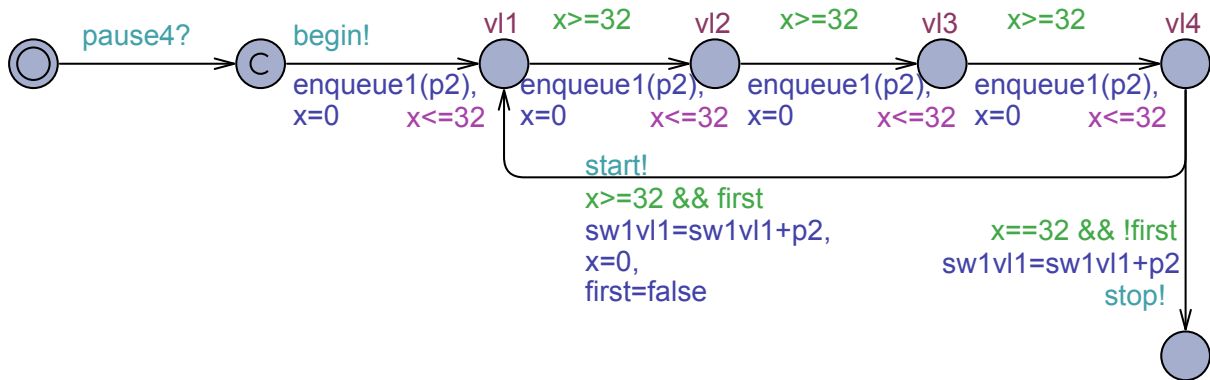
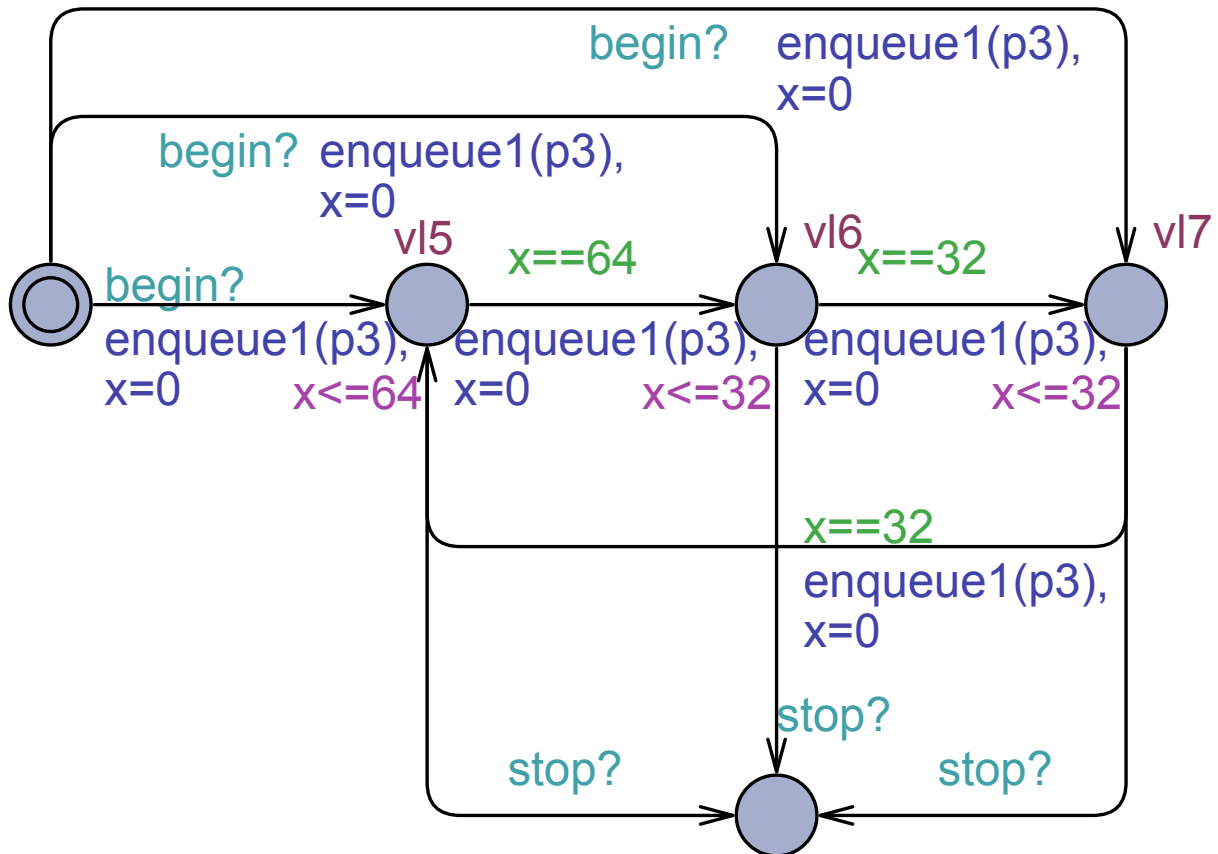


Figure 4.4 – Sequence of $e1$

- **Group G1:** For VLs emanating from the end system of the VL under study. In the example in Figure 4.3, G1 consists of TA of $e1$ which constructs the transmission sequence of $v1, v2, v3$ and $v4$.
- **Group G2:** For VLs starting from end systems connected directly to the same switch as the end system of VL under study. In the example in Figure 4.3, G2 consists of TA of $e2, e3, e4, e5, e6$ and sporadic VLs $v19, v20$.
- **Group G3:** For VLs joining VL under study from other switches. In the example in Figure 4.3, G3 consists of TA of $e7, e8, e9, e10$.

Periodic VLs emanating from the same end system are modelled within a single automaton to ensure offsets between them, while each sporadic VL is modelled in a separate automaton whether it originates from the same end system or not *e.g* $v19$ and $v20$ are modelled in separate automata. Modelling of each group is explained below. During this modelling, synchronization and buffers related variables are used which will be explained later in their respective sections.

Modelling of G1 Figure 4.5 shows the TA for $e1$. This TA starts with signal $pause4!$ and committed state ensures that it generates signal $begin!$ at the same time. Signal $pause4!$ and $begin!$ are used for synchronization as explained in table 4.2. VLs $v1, v2, v3, v4$ are generated by $e1$. States $v1$ to $v4$ represent corresponding VLs and invariant $x \leq 32$ represents offsets between periodic VLs. As an example, $v3$ offset is $32ms + 32ms$ as shown in table 4.1. $p2$ represents packet size. Here, $p2$ is equal to $2ms$ (transmission time equivalent to the size of the packet) but we can specify different packet sizes for each VL. Function $enqueue1(p2)$ stores an integer $p2$ in a FIFO array representing the output port of $S1$. For the VL under study, $v1$, we use a global integer $sw1v1$ instead of FIFO array. At the transition leading to state $v1$, $sw1v1$ is incremented by $p2$, modelling the transmission of a packet of $v1$ from $e1$ to switch $S1$ and a signal $start!$ is emitted so that measuring automaton, shown in figure 4.12, starts measuring time. Boolean $first$, initially set to *true*, is used to ensure that we stop the automaton after two complete hyper periods of $v1$ and generate synchronization signal $stop!$, to stop end systems from generating further frames.

Figure 4.5 – Timed automata of $e1$ Figure 4.6 – Timed automata of $e2$

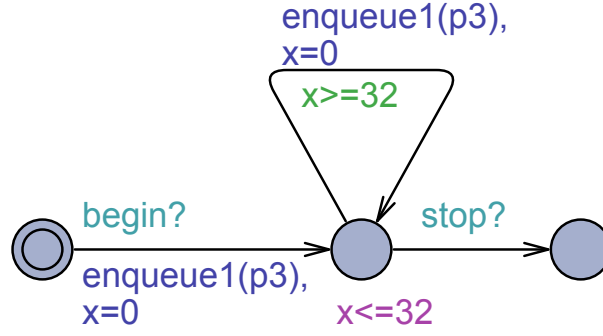
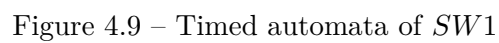


Figure 4.7 – Timed automata of sporadic VL

Modelling of G2 Automata of $e2$ to $e6$ build every possible output sequence corresponding to a scenario which is candidate for the worst-case end-to-end delay of $v1$ i.e which arrives synchronously with $v1$. Figure 4.6 shows the TA corresponding to the end system $e2$. It starts with synchronization signal *begin* generated by automaton of $e1$, and a frame from either $v5$ or $v6$ or $v7$ is transmitted thanks to non deterministic transitions from initial state of the automaton. Rest of the automaton works similar to automaton of $e1$. The automaton of $e3$, $e4$, $e5$ and $e6$ is similar to the one of $e2$ except for sporadic VLs $v19$ and $v20$. As discussed before, for worst case scenario, we can treat sporadic VLs as periodic VL with period equal to BAG but without any offset assignment. Therefore, sporadic VL is modelled as an independent end system with only one VL, as shown in Figure 4.7 for VL $v19$.

Modelling of G3 Figure 4.8 shows the automaton for end system $e9$. As mentioned before, we need to pause and resume this automaton for synchronization purpose, so we use integer variable $x1s$ as clock for this end system. As in automata of G2, sequence of $e9$ can start by sending a frame of either $v27$ or $v28$ or $v29$. Frame of these VLs are enqueued in switch $S3$ using the function $enqueue3(p3)$, where $p3$ represents the packet size. The sequences generated by $e9$ need to respect the offsets. This is modelled by condition on integer clock $x1s$ in the state as invariant. This automaton can be paused and resumed by two different signals (*pause3!* and *pause4!*). In each state, if a pause signal is received, the current value of clock integer $x1s$ is stored in a temporary integer tmp and automaton waits in a pause state. On resume signal, indicated by *go3!* or *go4!*, value of tmp is assigned back to clock integer $x1s$ and automaton can evolve. Automata of end system $e7, e8, e10$ are modelled on the same principle.



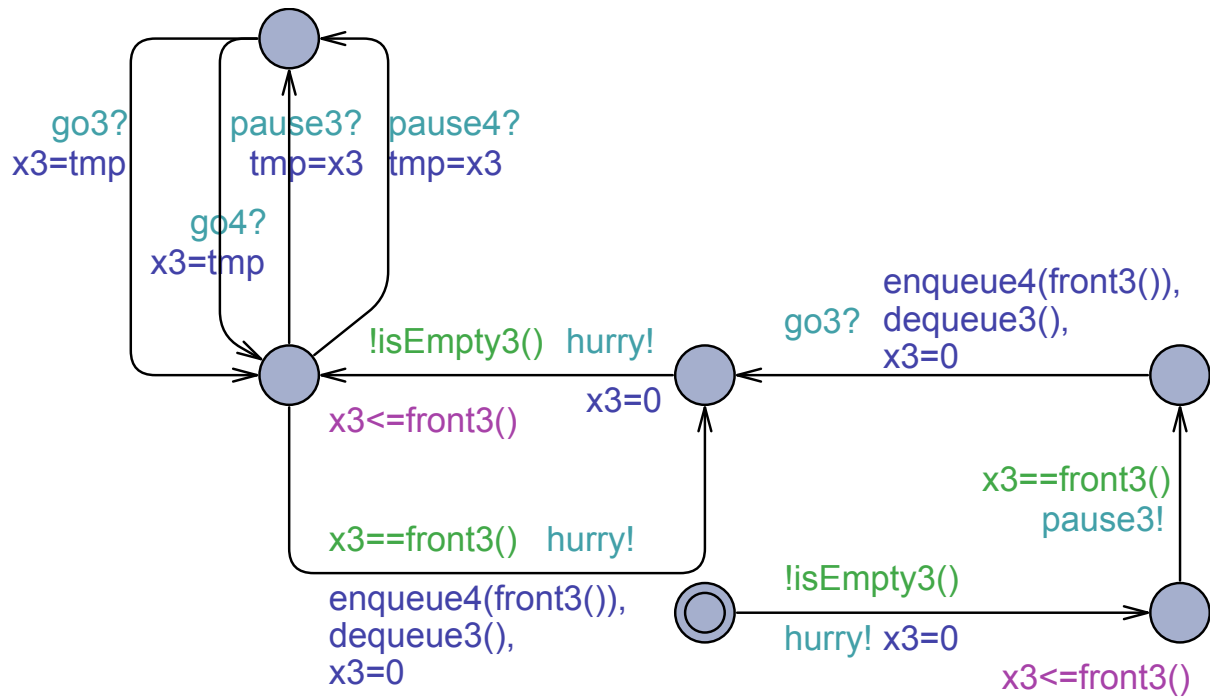
4.2.2 Modelling the Switches

The timed automata of the switch $S1$ is shown in Figure 4.9. For packets other than packet under study, switch output port is modelled as FIFO array. UPPAAL verification is very sensitive to size of arrays as discussed in [Discussion Group 2010]. Therefore, in this modelling approach, only one dimensional array is used to store packet size to implement switch output port FIFO queue as shown in code Listing 4.1. Switch input port is not modelled because transfer of packet from input port to output port has constant delay, which is represented as switching latency and can be added to overall end to end delay. The automaton of $S1$ leaves initial state as soon as the FIFO is not empty (meaning a packet has arrived) or the global variable for VL under study, $sw1vl1$, is greater than zero (meaning packet under study has arrived at switch $S1$). Signal *hurry!* is *urgent* and is used to ensure that the transition is taken immediately without any time lapse. Automaton waits in state *fifo* or *vlus* for the time equal to size of the packet. This behaviour models transmission time on AFDX link. After this time elapses, the packet is stored in the FIFO of next switch ($S5$) in the path, which in this case is done by function *enqueue5(front1())* and it is removed from FIFO of $S1$ by using *dequeue1()*. If the packet is from $v1$, then global variable for VL under study of next switch, in this case $sw2vl1$, is incremented by $sw1vl1$. This automaton also generates signal *go4!* as soon as it is ready to transmit first frame. *go4!* is then used by $e7$, $e8$, $e9$, $e10$, $S2$, $S3$ and $S4$ to start their automata.

Listing 4.1– Code for FIFO queue of output port of switch $S1$

```
intb list1 [20][1]; // columns: packet size
int [0, 19] len1; //points to first free place which is one more than last
    element in queue
void enqueue1(const int size) //store an element on top of queue (at last
    free position)
{
    list1[len1][0]=size;
    len1++;
}

void dequeue1() //remove top most element of the queue (at position 0)
{
    intb i = 0;
    len1 -= 1;
    while (i<len1)
    {
        list1[i][0] = list1[i+1][0];
        i++;
    }
}
```

Figure 4.10 – Timed automata of *SW3*

```

list1[i][0] = 0;
}

intb front1() //get front element id of queue (at position 0) (returning tx
time of packet )
{
    return list1[0][0];
}

bool isEmpty1() //test of queue is empty or not
{
    return (len1==0); //(list[0]==-1)
}

```

The timed automata of the switch *S3* is shown in Figure 4.10. It is modelled on the same principle as *S1* except that it can be paused/resumed by two signals: *pause3!* and *pause4!*. It leaves initial state as soon as the FIFO is not empty (meaning a packet has arrived). After time equal to packet size elapses, the packet is stored in the FIFO of next switch (*S4*) in the path, which in this case is done by function *enqueue4(front3())* and it is removed from FIFO of *S3* by using *dequeue3()*. Whenever signal *pause3!* or *pause4!* is received, current value of integer clock *x3* is stored in a temporary integer *tmp* and restored back to *x3* on resume signal *go3!*

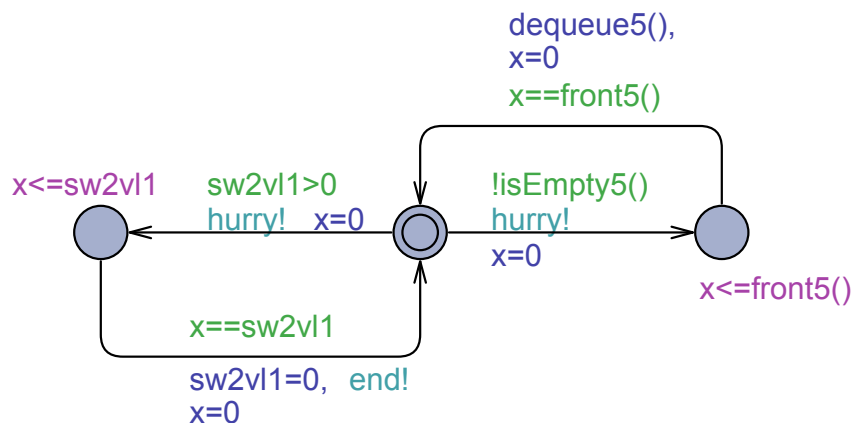


Figure 4.11 – Timed automata of *SW5*

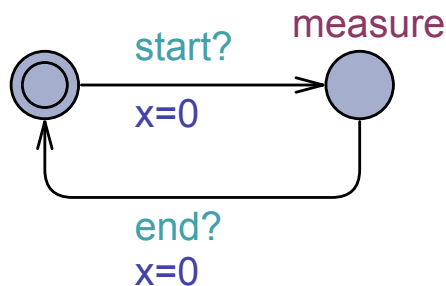


Figure 4.12 – TA for the measurement

or *go4!*. This automaton also generates signal *pause3!* as soon as it is ready to transmit first frame. *pause3!* is then used by *e7* and *e8* to start their automata. *SW2* and *SW4* are modelled in similar way.

The timed automata of the switch *S5* is shown in Figure 4.11. It is modelled in same way as *SW1* except that a signal *end!* is generated after waiting for time equal to global variable for VL under study *sw2vl1*. *end* signal represents reception of frame under study at destination end system and stops all automata of the network.

4.2.3 Modelling the Synchronization

A synchronisation mechanism is needed between the different TA in order to ensure that only those scenarios are considered which are candidate for the worst case. The idea is to synchronize the input sequences of each switch. Considering the example in Figure 4.3, it comes to process, in a data flow way:

t	G1	G2	G3		G4				
	(e1)	(*)	(e7, e8)	(e9, e10)	(S1)	(S2)	(S3)	(S4)	(S5)
t_0	pause till <i>pause4!</i>	pause till <i>begin!</i>	pause till <i>pause3!</i>	emit 1st packet for S3	ready for packet	ready for packet	ready for packet	ready for packet	ready for packet
t_1							ready to emit packet for S4. broad- cast <i>pause3!</i>		
t_1			start with <i>pause3!</i> . emit 1st packet for S2	pause till <i>go3!</i>			pause till <i>go3!</i>		
t_2						ready to emit packet for S4. broad- cast <i>go3!</i>			
t_2				resume with <i>go3!</i>			resume with <i>go3!</i>		
t_3								ready to emit packet for S5. broad- cast <i>pause4!</i>	
t_3	start with <i>pause4!</i> . broad- cast <i>begin!</i>		pause till <i>go4!</i>	pause till <i>go4!</i>		pause till <i>go4!</i>	pause till <i>go4!</i>	pause till <i>go4!</i>	
t_3	emit 1st packet for S1	start with <i>begin!</i> . emit 1st packet for S1							
t_4					ready to emit packet for S5. broad- cast <i>go4!</i>				
t_4			resume with <i>go4!</i>	resume with <i>go4!</i>		resume with <i>go4!</i>	resume with <i>go4!</i>	resume with <i>go4!</i>	
t_5	stop after 2 hyper period. broad- cast <i>stop!</i>								
t_5		stop with <i>stop!</i>	stop with <i>stop!</i>	stop with <i>stop!</i>					

(*)e2, e3, e4, e5, e6, v19, v20

Table 4.2 – Synchronization among different groups of VLs.

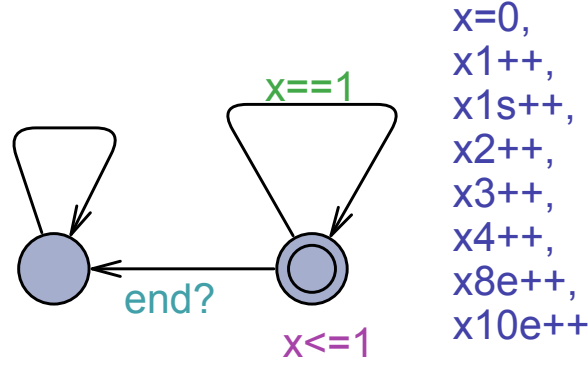


Figure 4.13 – TA for integer clocks

- $e9$ and $e10$ synchronize at $S3$ and wait for $S2$ where $e7$ and $e8$ synchronize,
- $S2$ and $S3$ synchronize at $S4$ and wait for $S1$ where $e1$ to $e6$ synchronize,
- $S1$ and $S4$ synchronize at $S5$.

The implementation of this synchronization mechanism is summarized in Table 4.2. End systems $e9$ and $e10$ start building frame sequence for switch $S3$ and pause as soon as $S3$ is ready to transmit the first frame. This is indicated by the broadcast signal *pause3!*. Then $S3$ waits for the first frame from switch $S2$. End systems $e7$ and $e8$ start building frame sequences as soon as they receive the broadcast signal *pause3*. As soon as $S2$ is ready to transmit a first frame, $e9$, $e10$ and switch $S3$ are resumed with signal *go3!*. This ensures that a frame from $S2$ arrives at the same time as a frame from $S3$ at $S4$. As soon as $S4$ is ready to transmit a first frame, switch $S4$ pauses $e7$, $e8$, $e9$, $e10$, $S2$ and $S3$ with signal *pause4!*, in order to wait for a frame from $S1$. Finally, end system $e1$ broadcasts signal *begin!* as soon as it receives signal *pause4!*. Thus, $e1$ starts building sequence at the same time as end systems $e2$, $e3$, $e4$, $e5$, $e6$ and sporadic VLs $v19$, $v20$ (group G2). Switch $S1$ starts building sequences as soon as it receives a frame and it resumes other paused automata with signal *go4!*. This ensures that a frame from $S1$ and $S4$ arrive at the same time at $S5$.

4.2.4 Utility Automata: modelling of the buffers

In the preliminary Timed Automata approach presented in [Adnan 2011a], FIFO buffers in the output port of switches are modelled by *Arrays*. Each element of an array corresponds to a frame which is waiting for the output link associated with the port. A frame is described by its transmission duration and the identifier of its flow. Such a modelling does not fit well with

UPPAAL, which is very sensitive to the size of arrays [Discussion Group 2010]. Consequently, the modelling proposed in the present section limits the size of the arrays. In order to do that, each element of the array only contains the transmission duration of the corresponding frame (the flow identifier is no more used).

In [Adnan 2011a], the flow identifier was used in order to differentiate three kinds of frames:

- The frame under study; it is mandatory to know when this frame reaches its destination in order to stop the measurement of time.
- The frame which will leave before the destination; they will not be stored in the FIFO buffer of the next switch.
- All other frames; they also go to the destination end system.

Since we don't have the flow identifier of a frame in the array anymore, we don't know to which VL a given frame in an array belongs to. In order to solve this problem, only the frames of the third kind are stored in the arrays. The frame under study (first kind) is modelled by using integers ($sw1vl1, sw2vl1$ in the example). When a switch detects that the corresponding integer is not null, it knows that the frame under study has arrived and has to be transmitted. Similarly, the frames of second kind are also modelled by using integers.

4.2.5 Utility Automata: end to end delay computation

Figure 4.12 shows the automaton to measure end-to-end delay. It starts with signal *start!* and stop measuring end to end delay when $v1$ reaches $e11$, indicated by signal *end!* in automaton of switch $S5$. For VLs joining VL under study $v1$ from other switches, it is necessary that they are synchronized in such a way that at any switch output port, packets arrive at the same time as $v1$ even after crossing different switches. This requires that we store the current values of *clock* variables when we pause the automaton and assign these values to *clock* variables on resume. But UPPAAL does not provide such feature for its *clock* variables. Therefore, we modelled an automata which increases integer variables every clock cycle as shown in Figure 4.13 and use them as "integer clocks". Such an implementation has advantage that we can read and modify clock values and also reduce the number of clock variables in order to reduce state-space. The automata of VL under study and switches it crosses are never stopped, hence these automata don't use integer clocks. Figure 4.13 shows the automaton for integer clocks. UPPAAL clock variable x is used to increment all integers by one, after each cycle of x . The automaton is stopped when packet under study has reached destination indicated by signal *end!*.

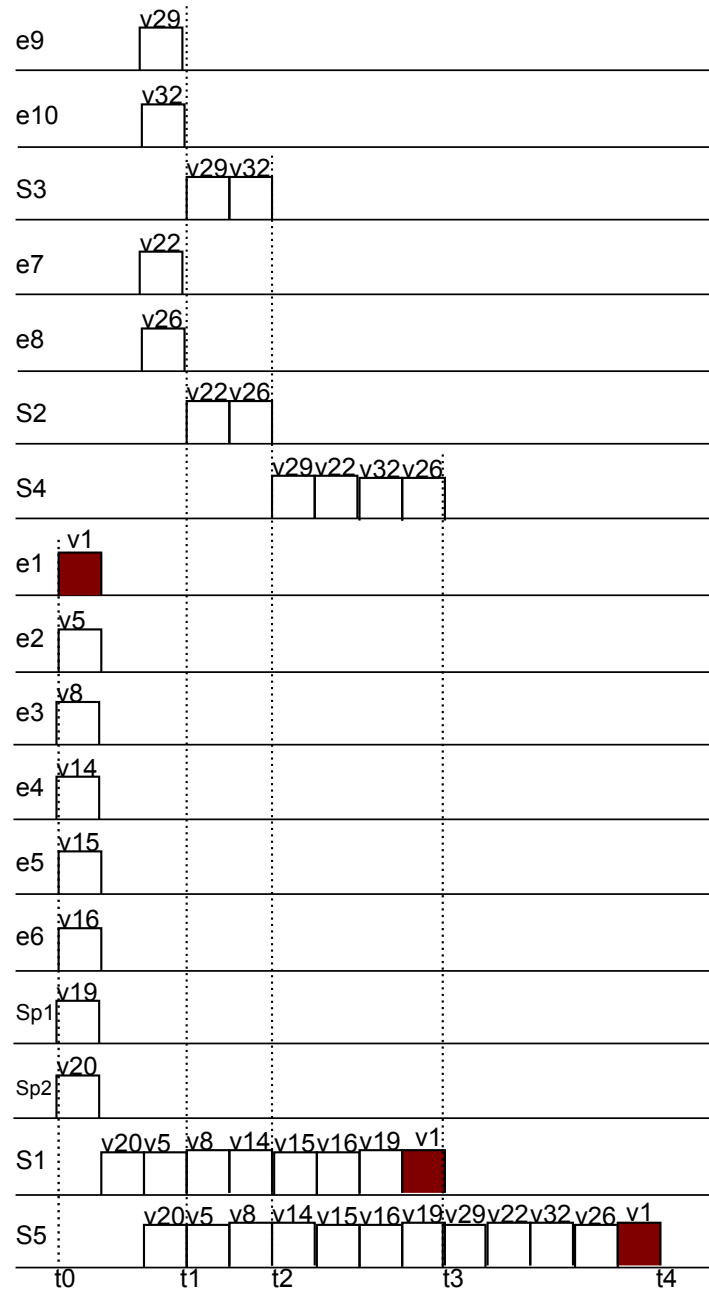


Figure 4.14 – A worst case scenario for v_1

The worst-case delay for the VL under study is the smallest value d such that the value of clock x in the state *measure* is always less than or equal to d . This is obtained by querying the timed automata using the following CTL formula:

$A[] ((E2E.measure \text{ imply } x \leq v))$

This formula evaluates to *true* when there is no scenario leading to $x > v$. It evaluates to false otherwise. Thus the worst-case delay for the VL under study corresponds to the lowest value of v for which above formula evaluates to true. A corresponding worst-case scenario is obtained by considering the highest value of v for which the formula evaluates to false. Indeed, UPPAAL generates a trace of one scenario which leads to this state. As an example, the worst-case delay for VL $v1$ in Figure 4.3 is $26ms$ and a corresponding scenario is depicted in Figure 4.14.

4.3 Limits of the approach

The modelling proposed in this chapter allows the analysis of the AFDX configuration in Figure 4.3 including 32 VLs. It is computed in 10 minutes on a PC with 3.3 GHz Intel Core 2 Duo processor using 2 GB RAM. Configurations with more VLs cannot be analyzed within a reasonable time. As compared to the version of TA approach discussed in [Adnan 2011a] (18 VLs), the upper limit of this version is 32 VLs and it also supports sporadic VLs. Thus, the approach presented in this model brings a clear improvement. This work was published in [Adnan 2012].

In principal, timed automata models are based on *regions* and *zones* (*zone* is a convex union of regions) [Bérard 2001]. The graph based on these *zones* leads to more states and *regions* as compared to the Java based approach. As an example, consider a simple timed automata shown in figure 4.15. This automata represents a simple transition from a state q to a state r depending on values of clock $x1$ and $x2$. This can be a representation of a packet transmission by an end system of an AFDX network. The region graph of this simple automata will comprise of about 30 states and 50 regions. A part of this graph is shown in figure 4.16. On the other hand, if we want to represent same transmission of one packet from an end system using Java based tool, it can be represented by a single scenario.

Another limiting factor in timed automata based approach is the inherent complexity of the algorithms used for model checking the timed automata. The number of *regions* of a given timed automata model grow exponentially with the number of clocks. For n clocks and constraints in which every constant k is upper bounded by M , the complexity of number of regions is $\mathcal{O}(n!M^n)$, as determined by Bérard Béatrice et al. in [Bérard 2001].

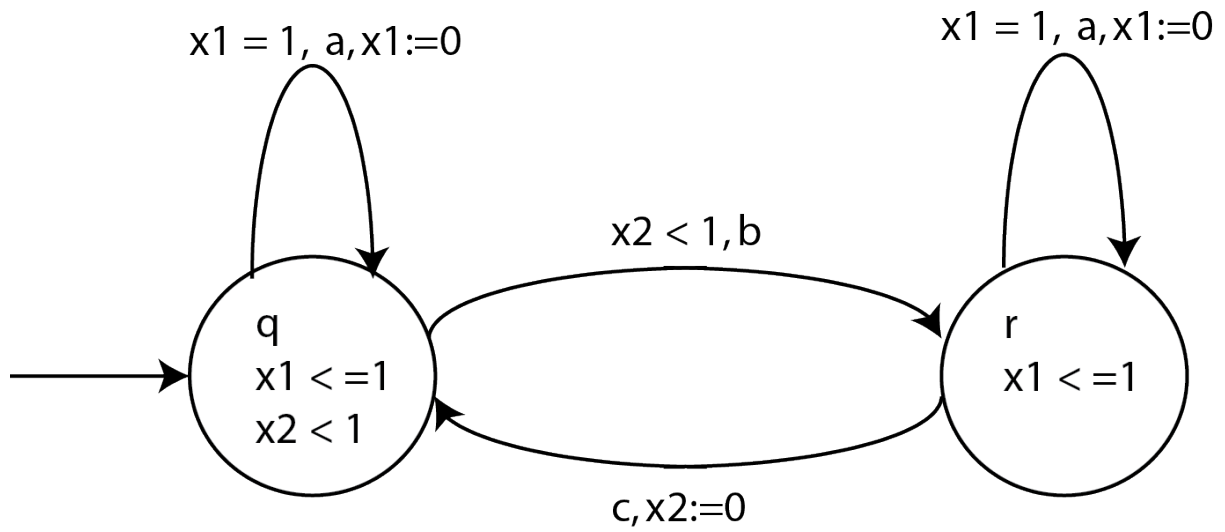


Figure 4.15 – A simple timed automata.

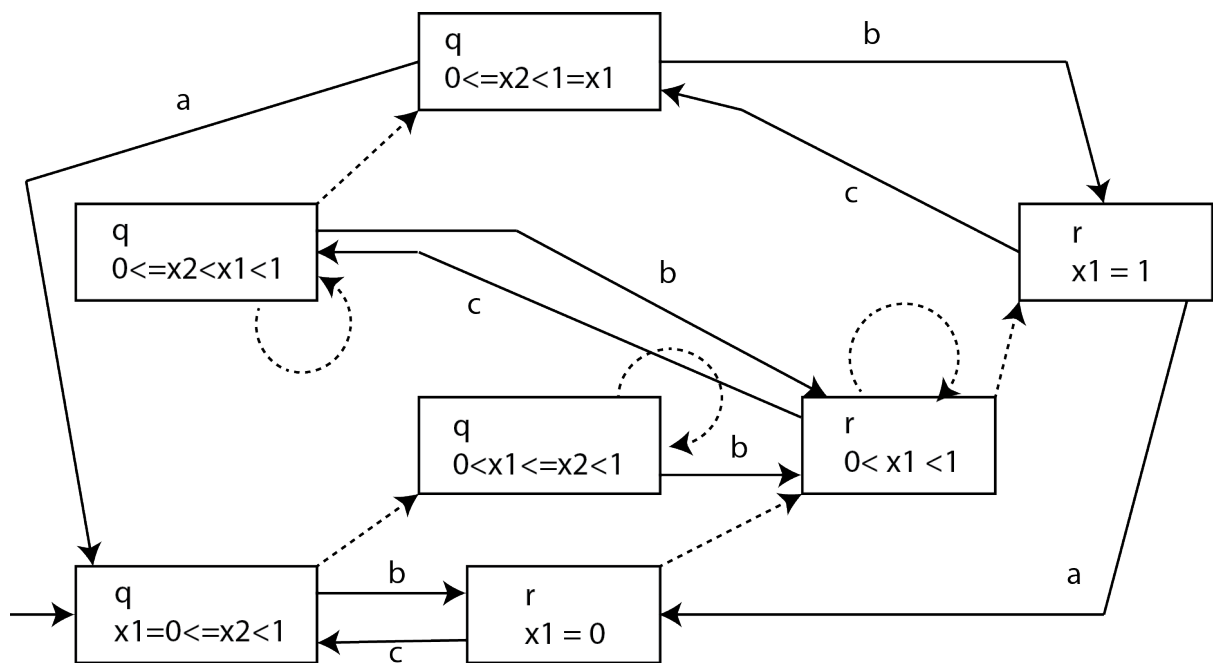


Figure 4.16 – Partial zone graph of the simple timed automata.

4.4 Conclusion

This chapter presented an improved timed automata approach using timed automata for computing exact worst-case delays of AFDX periodic and sporadic flows. The method have been encoded into UPPAAL model checker. This approach is based on a drastic reduction of the number of scenarios which are candidate for the worst-case end-to-end delay of a given flow. The size of the network configurations which can be analyzed by this new approach, upto 32 VLs, is significantly larger than what can be done with the previous approach in [Charara 2006a]. On the other hand state-space reduction requires the exploitation of AFDX network properties, in order to consider only those cases that may be candidate for the worst case scenario. This means that the approach will be very specific to AFDX network. Though, the methodology and ideas behind it can be extended to other domain and problems as well.

For a given computational capacity, timed automata based approach is limited for an industrial configuration of the AFDX network. Nevertheless, the timed automata based approach is still interesting and of a great value. Timed automata is based on formal methods, hence it allows us to verify properties which ensure that the model always behave correctly.

The proposed approach is not limited to only the AFDX network but can cope with any switched Ethernet network. It can also be used for analysis of different scheduling techniques in order to compute delay bounds in multi-processor networked architectures. Finally, it could be extended to switched Ethernet networks with service disciplines other than FIFO [Zhang 1995].

Due to the complexity of this modelling approach it is clear that with given computational resources it will be very hard to analyze an industrial configuration of the AFDX network. But this method allowed us to better understand the behaviour of an AFDX network and to define the Critical Instance property which allows to compute the maximum waiting delays in the FIFO queues of the switches. Considering this, in Chapter 5, we propose a new approach to compute the worst-case end to end transmission delays based on the generation of sequences of VL which are candidate for the worse-case end to end delays.

A New Approach Based on Exhaustive Simulation to Compute the Exact Worst-Case End to End delays

Contents

5.1	Modelling of the network and a scenario	84
5.1.1	Nomenclature and definitions	85
5.1.2	Modelling of a scenario	86
5.1.3	Reducing the number of scenarios	87
5.2	Computing worst case end to end delays using sequences	88
5.2.1	Computation of delay and merging of sequences at a switch output port . .	89
5.3	Worst-case end to end delay computations on a simple AFDX network using sequences	90
5.3.1	Presentation of the system	90
5.3.2	Computing the worst case end to end delay of VL under study	90
5.3.3	Computation of the sequences generated at the input of switch $S2$	92
5.3.4	Computation of the resulting sequences at the output of switch $S2$	93
5.3.5	Computation of the sequences at the input ports of switch $S1$	94
5.3.6	Computation of the sequences at the output of switch $S1$	95
5.3.7	Computation of the sequences at the input ports of switch $S3$	96
5.3.8	Computation of the sequences at the output of switch $S3$	97
5.4	Evaluation of the sequence based approach	97
5.5	More Improvements and reduction in scenarios	99
5.5.1	Modeling of Sporadic traffic	101
5.5.2	Further reduction of scenarios	102

VL	BAG(ms)	Size(ms)	path
v1	32	5	e1-S1-S3-e8
v2	32	2	e1-S1-e5,e1-S1-S3-e7
v3	32	3	e2-S2-S1-S3-e8
v4	32	3	e2-S2-e4,e2-S2-S1-S3-e7
v5	32	2	e3-S2-S1-e5
v6	32	3	e3-S2-e4,e3-S2-S1-S3-e7
v7	32	3	e6-S3-e8
v8	32	3	e6-S3-e7

Table 5.1 – AFDX network configuration data

5.5.3	Candidate scenario for worst case delays	107
5.5.4	Algorithm to further reduce number of cases	107
5.6	Conclusion	111

In the Chapter 4, the timed automata based models can be used to find exact worst-case end to end delays for AFDX network. The problem with that approach is the use of a standard model checker which is not well adapted for specific task of AFDX network end to end communication delay computations. This chapter proposes a new approach based on Exhaustive Simulation to calculate exact worst case end-to-end delays. Exhaustive simulation is similar to model checking in the sense that both approaches consider all possible cases or scenarios. The idea is to compute the sequences of frames of the VLs transmitted by each end system and to consider only those sequences which are candidate for the worst-case end to end delays. To reduce the search space, properties of the AFDX network are considered as the *Critical Instance* property shown in Chapter 4. A tool has been developed for this purpose and the exact worst-case end to end delays of an AFDX network has been computed by using this tool which will be presented in Chapter 6.

5.1 Modelling of the network and a scenario

In the following sections we will explain the modelling methodology we adopted to carry out the end to end delay computations.

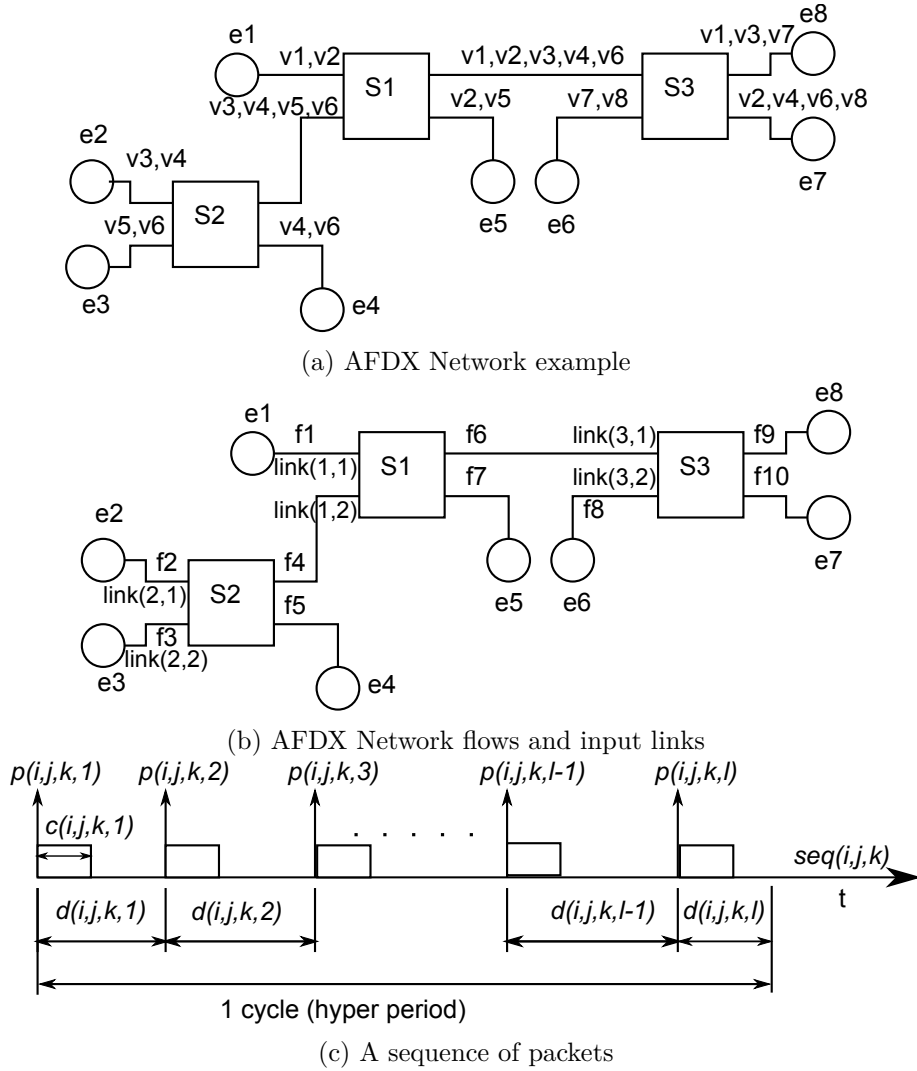


Figure 5.1 – AFDX Network architecture.

5.1.1 Nomenclature and definitions

In this chapter, we use certain notations to define an AFDX network. We will define these notations and other terminologies with the help of an AFDX network shown in figure 5.1a. This example network consists of 3 switches $S1, S2, S3$, and eight end systems $e1$ to $e8$. Each end system generates a set of VLs. There are total of eight VLs named as $v1$ to $v8$. The characteristics and paths of these VLs are shown in table 5.1. Size of the VL in the table 5.1 is shown in terms of transmission time of the packet assuming 100 MHz link speed. These times are purposely chosen as larger than the normal packet sizes in order to show that worst case on each port does not always lead to overall worst case and will be demonstrated later in section

5.3.

Each link in an AFDX network carries *flows* of packets, named as $f1$ to $f10$ in figure 5.1b. Each switch in the network has at least one input link and at least one output link. The input links of a switch Si are denoted as $link(i, j)$ where i represents the switch number and j represents j^{th} input link of this switch, as shown in figure 5.1b. If there are m_i total input links of switch Si then $1 \leq j \leq m_i$. This terminology is used only for input links of a switch. For all output links, we use *flow* names.

A flow coming out of an end system consists of an ordered list of packets. The order of these packets of the flow is known for strictly periodic VLs. For sporadic VLs, this order can have many possible permutations. We call one possible order of packets as a *sequence*. The number of unique sequences generated by each end system depends upon the number of VLs of the end system, their BAG values, and scheduling being used for periodic VLs. A set of sequences that can be received at input link $link(i, j)$ of a switch Si is denoted as $seq(i, j)$ where i is the switch number and j is the j^{th} input link of the switch and $1 \leq j \leq m_i$. Each individual sequence among the set $seq(i, j)$ is denoted by $seq(i, j, k)$ where k represents k^{th} sequence of the set.

Packets of a particular sequence $seq(i, j, k)$ have defined order and temporal relation among them. Each packet of the sequence $seq(i, j, k)$ is denoted as $p(i, j, k, l)$ where l represents l^{th} packet of the sequence $seq(i, j, k)$. Temporal relation between two consecutive packets is defined by a gap between the packets and is denoted as $d(i, j, k, l)$ which represents the distance of packet $p(i, j, k, l)$ from the next packet in the sequence in terms of transmission time on the link. Finally, the transmission time of a packet $p(i, j, k, l)$ is denoted as $c(i, j, k, l)$. This is illustrated in figure 5.1c.

5.1.2 Modelling of a scenario

As discussed in section 4.1.1, a scenario is one possible case for the end to end communication delay analysis. In the context of this chapter, a scenario represents one possible permutation of sequences from each link of the network. This section describes how to model a scenario in the model that we will use for exhaustive simulation in this chapter. A sequence of periodic frames $seq(i, j, k)$ is transmitted on each link $link(i, j)$ of switch Si where $1 \leq i \leq n$ and $1 \leq j \leq m_i$. The sequence is cyclic (repeating itself over an hyper period P_h) and is defined by a circular list, as shown in figure 5.1c. The horizontal distance between the l^{th} packet $p(i, j, k, l)$ of $seq(i, j, k)$ and its successor is derived from the scheduling offsets. This distance is denoted $d(i, j, k, l)$. Each packet $p(i, j, k, l)$ has a transmission time $c(i, j, k, l)$ where $d(i, j, k, l) \geq c(i, j, k, l)$ meaning

that a packet arrives at a switch after at least its transmission time. This phenomenon is called serialization or grouping in previous works [Charara 2006a, Fraboul 2002a, Bauer 2010, Martin 2006a] using network calculus and trajectory approach.

5.1.3 Reducing the number of scenarios

The approach proposed in this chapter is based upon the assumption that the worst case at a given output port happens when the packet under study arrives at the output port at the same instant as one packet from all the other input links of the corresponding switch, as proved in section 4.1.2 in *Critical Instance* property. Now we present some other properties that will help us in modelling a scenario and in reducing number of scenarios which are candidate for worst case end to end delays.

Property 2: Periodicity of the outgoing sequence at an output port.

The sequence transmitted at the output port of Si is periodic if: (1) The sets of input sequences $seq(i, j)$ ($1 \leq j \leq m_i$) are periodic. (2) The output port is not overloaded, *i.e* the incoming traffic does not exceed the capacity of the output port.

Proof: The load of a sequence $seq(i, j, k)$ is defined as the ratio between the sum of the transmission times of all its packets and its hyper period P_h . The load L is defined as:

$$L = \frac{\sum_l(c(i, j, k, l))}{P_h}$$

where i, j and k are fixed and l varies from 1 to last packet of the sequence. If the load L is not greater than 1 then, during the hyper period P_h , there is a time when there is no backlog and the link is idle. Assuming that the sequences on all m_i input links of Si are periodic, let's define $P(i, j, k)$ as the period of the sequence $seq(i, j, k)$ transmitted on $link(i, j)$ and $P_h(i)$ the least common multiple of all the $P(i, j, k)$ ($1 \leq j \leq m_i$). Then the sequence of frames received at Si is periodic with period $P_h(i)$. Figure 5.2 illustrates $P_h(i)$ and the $P(i, j, k)$.

All the packets arriving from an input link of Si are transmitted in the output link of Si with a first in first out (FIFO) policy. In cases where the output link is not overloaded, there is at least one instant during the hyper period $P_h(i)$ with no backlog. Let's call t_{nb} such an instant. Then

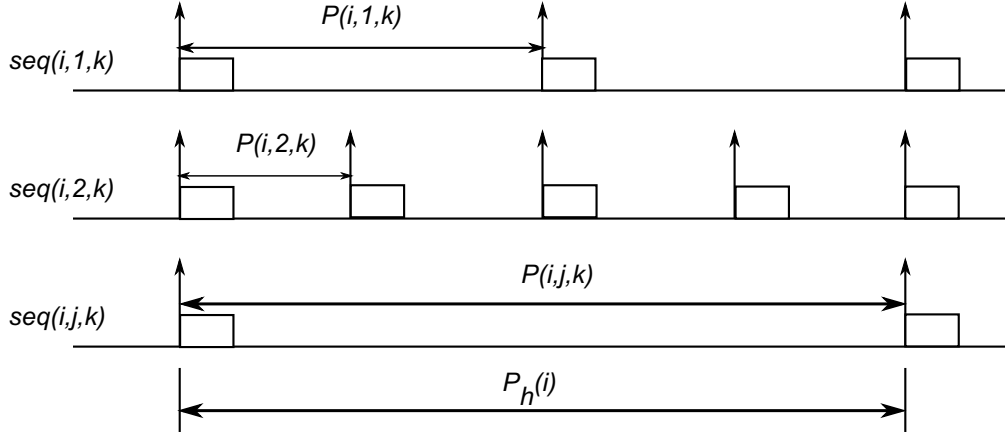


Figure 5.2 – Property 2.

$t_{nb} + zP_h(i)$ are also instants with no backlog, where z is an integer greater than zero. Since we have the same sequence of frames arriving at S_i in each interval $[t_{nb} + zP_h(i), t_{nb} + (z+1)P_h(i)[$, each of these sequences will generate the same transmission of frames at the considered output port. Consequently, the sequence of frames transmitted in the output port of S_i is periodic with period $P_h(i)$. ■

5.2 Computing worst case end to end delays using sequences

In order to compute the worst case end to end delays using sequences, the proposed approach is to compute the combinations of sequences which lead to the worst case scenario by considering the two properties given before. For this purpose, an algorithm has been developed and composed of two parts:

1. Calculations of backlog and combination of sequences on a local port at switch level
2. Global management of data flow based computations on network level.

On a local port level, each end system's packet generation sequence is stored as an ordered circular list (packets are periodically sent). Each entry of the list represents a packet with its characteristics (size, time of generation, scheduling offset etc). For a given output port, the computation is done according to the algorithm 1. Calculations of backlog and combination of sequences on a local port at switch level is done in step 3 and 4. Global management of data flow based computations on network level is done in step 1,2 and 5. The algorithm uses ordered

Algorithm 1 Worst case delay calculation on output ports

-
- 1: **while** all output ports have not been computed **do**
 - 2: select an output link for which the set of sequences of all the input links are already computed.
 - 3: compute the set of output sequences of this output link.
 - 4: For each sequence in the set, compute the backlog at the reception of the last packet of the sequence, which is packet under study. This backlog represents delay at this port for this sequence.
 - 5: **end while**
 - 6: The worst case delay for frame x at the destination output port is the largest sum of delays on each port passed by the VL in the set of generated sequences.
-

lists of packets sent by all the end systems which pass through the output port. The result of the algorithm is the worst case delay for the frame under study at the output port and all possible sequences of packets of all VLs sharing this output port of switch Si .

5.2.1 Computation of delay and merging of sequences at a switch output port

The process of calculating the backlog for a given scenario at an output port is illustrated in figure 5.3. We shift packets to align the arrival times of the required packets and then calculate backlog in this scenario, which is simply the time at which the last packet in the merged sequence finishes its transmission. Assuming packets p15, p25 and p34 arrive together from es1, es2 and es3 respectively, we shift all other packets according to their generation time and then build a new sequence from these three input sequences after they have passed from the output port. We take the first flow, in this case *es1*, and construct an array of packets with their arrival time and the time when they will finish their transmission at the output port considering this is the only flow at the port. Next we consider the second flow, in this case *es2*, and put its packets in the array. Since there are already packets of the first flow in the array, if there is an overlap of packets, we insert the packet before an existing packet if it arrives earlier, or after an existing packet if it arrives latter. If the packets do not overlap, we insert a new row in the table with packet's arrival time and transmission time. We repeat this process for all input flows. At the end, the array contains the resulting sequence of packets at given output of the switch.

At network level, this algorithm follows a data flow approach: calculation starts at source nodes initializing the set of sequences transmitted from each end system (where all the required data is already available) and continues towards the destination when all required data is available for each output port. The set of sequences computed at a given output port of a given

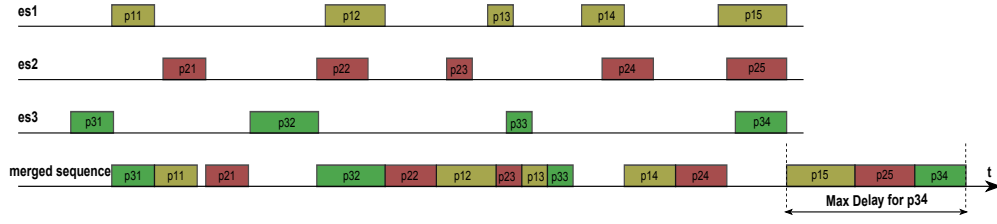


Figure 5.3 – Merging packet sequences and backlog calculation.

switch by the algorithm described earlier is used as input to the next switch (or destination node) in the path of the VL under study. The overall process is explained with an illustrative example in the next section.

5.3 Worst-case end to end delay computations on a simple AFDX network using sequences

5.3.1 Presentation of the system

In order to explain how the algorithm works, we will consider a model avionics system of an aircraft as shown in figure 5.4. This relatively small avionics system consists of three AFDX switches and eight end systems. This avionics system can be mapped to the block diagram of example of figure 5.1a with $e1$ as Flight Management System, $e2$ as Control Display Unit, $e3$ as Engine Controls, $e4$ as Flight Controls, $e5$ as Air Data and Inertial Reference Unit, $e6$ as Navigation and Radar, $e7$ as Engine/Warning Display and $e8$ as Primary Flight Display. Hence we can reuse the Table 5.1 for the configuration data for this AFDX network. The packet sizes of the VLs in the table 5.1 are larger than what AFDX standard allows for a single packet. This is chosen on purpose in order to demonstrate the fact that worst case delay on each port does not always lead to end-to-end worst case delay.

5.3.2 Computing the worst case end to end delay of VL under study

VL $v3$ is under study, which is a control message sent from Control Display Unit (CDU) to primary Flight Display (PFD). It originates at end system CDU, corresponding to end system $e2$ of figure 5.1a and follows the path $e2 - S2 - S1 - S3 - e8$. Following steps are needed to compute worst case end to end delay for VL $v3$:

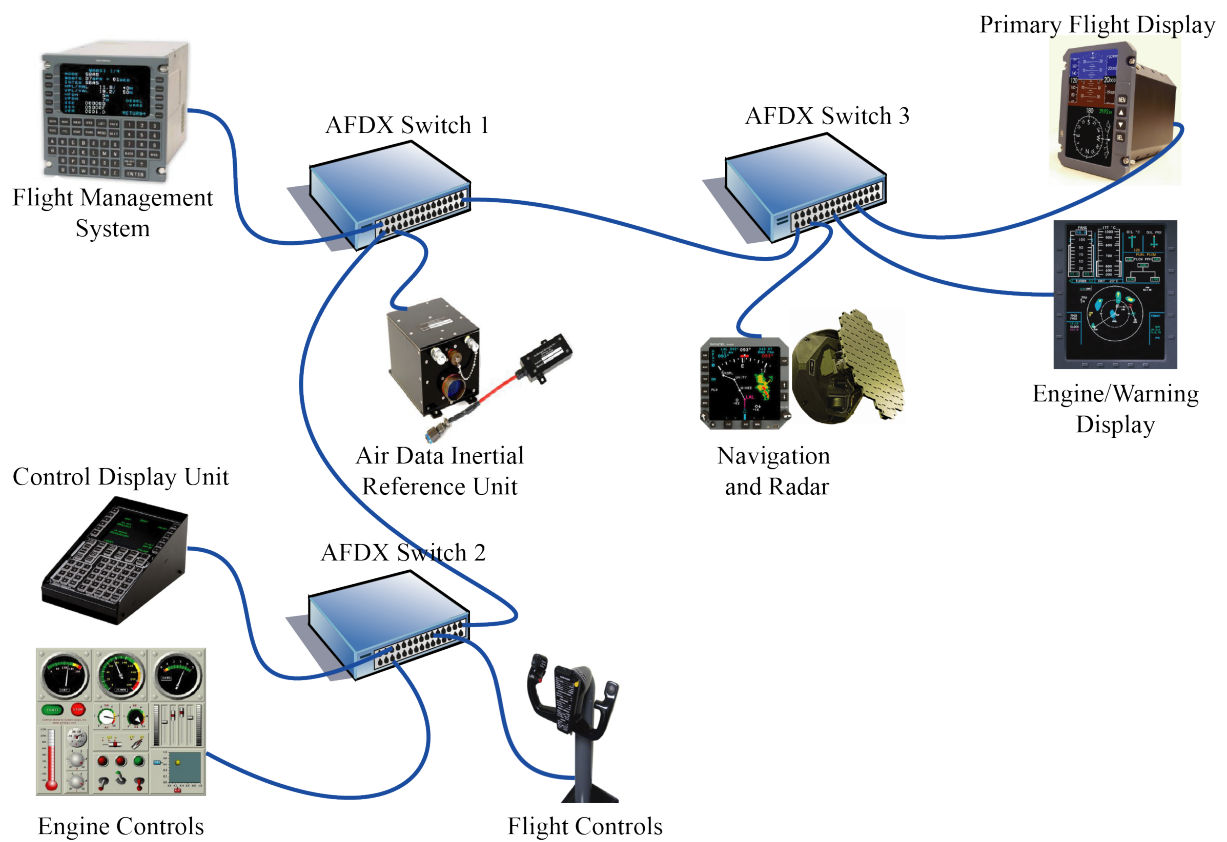


Figure 5.4 – A simple AFDX Network of a small aircraft.

1. Computation of the sequences generated at the input of switch $S2$.
2. Computation of the resulting sequences at the output of switch $S2$.
3. Computation of the sequences at the input ports of switch $S1$.
4. Computation of the sequences at the output of switch $S1$.
5. Computation of the sequences at the input ports of switch $S3$.
6. Computation of the sequences at the output of switch $S3$.

5.3.3 Computation of the sequences generated at the input of switch $S2$

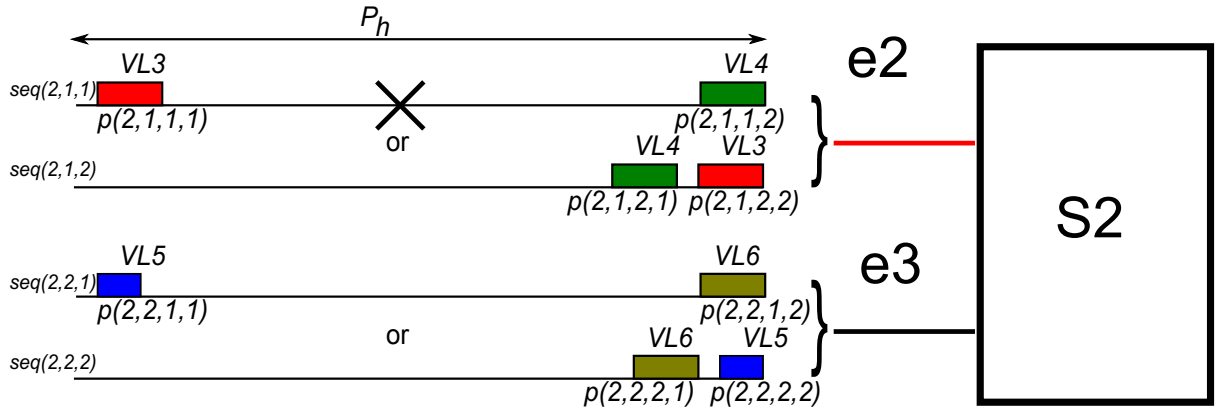


Figure 5.5 – Sequences generated at input of Switch $S2$.

Let us assume that the end system $e2$ schedules its VLs in such a way that there is no waiting delay at the end system output port. Hence the worst case delay for all the VLs at the end system is just the transmission time of the largest VL packet, i.e $3ms$ for $v3$ at end system $e2$. Figure 5.5 illustrates the sequences of frames transmitted from $e2$: $seq(2,1,1)$ and $seq(2,1,2)$ on link $link(2,1)$ and sequences of frames transmitted from $e3$: $seq(2,2,1)$ and $seq(2,2,2)$ on link $link(2,2)$.

5.3.4 Computation of the resulting sequences at the output of switch S_2

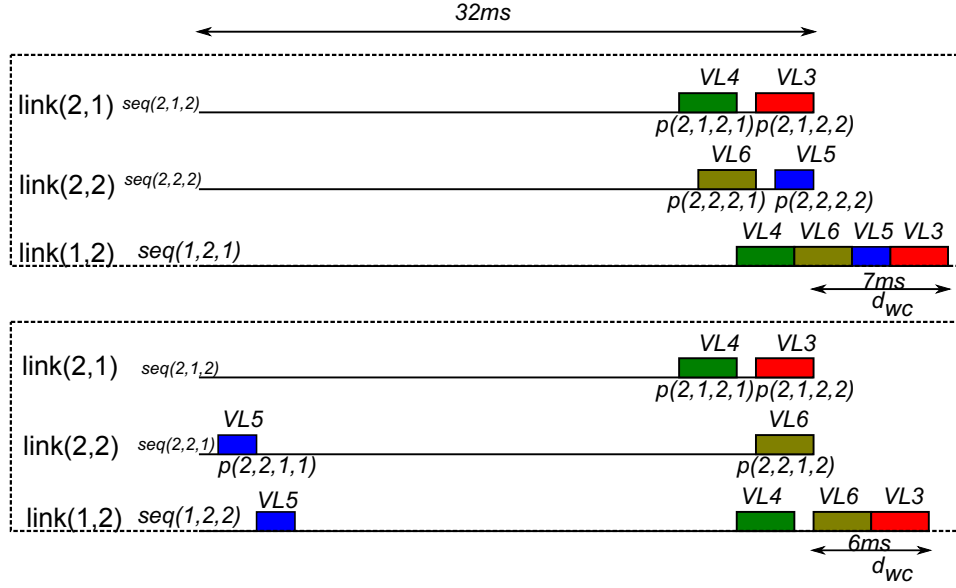


Figure 5.6 – Construction of sequences at output of Switch S_2 Port 1.

The next hop in path of VL v_3 is switch S_2 output port. At this port, VL v_3 , v_4 , v_5 and v_6 compete for the link $link(1,2)$. Therefore we need to construct sequences $seq(2,1,1)$ for VL v_3 , v_4 and $seq(2,2,1)$ for VL v_5 and v_6 on links $link(2,1)$ and $link(2,2)$ respectively before we can proceed further. Figure 5.5 illustrates these sequences. As we described earlier, this algorithm checks all possible combinations of packets in the sequences. Hence the total number of sequences generated at the output port of a switch can be calculated by multiplying number of packets in a sequence in each input link of the switch. In this case there are two input links ($link(2,1)$ and $link(2,2)$) of switch S_2 , each having two packets. Hence the total number of sequences generated at switch S_2 output port for link $link(1,2)$ is calculated to be $2 * 2 = 4$. But according to Property 1, we are only interested in sequences where packet under study is transmitted in the end, which in this case is packet $p(2,1,2,2)$. Therefore out of 2 possible sequences generated from send system e_2 , we only consider one sequences: $seq(2,1,2)$. This results in total of $1 * 2 = 2$ sequences which can be candidate for worst case delays, which is a reduction of two sequences (50%) at the input of switch S_2 .

To generate these sequences, we first consider the case where packet $p(2,1,2,2)$ and packet $p(2,2,2,2)$ arrive together at switch S_2 output port. Figure 5.6 shows this scenario as $seq(1,2,1)$. The same process is repeated for packets $p(2,1,2,2)$ and packet $p(2,2,1,2)$ (2 possible permutations). At the end we obtain 2 sequences $seq(1,2,1)$ and $seq(1,2,2)$ for link $link(1,2)$. The worst case delay for packet $p(2,1,2,2)$ of VL v_3 is in $seq(1,2,1)$. The backlog

size for each generated output sequence is the maximum number of adjoining packets. This backlog size can be used to bound the FIFO buffer depth. In this example, the largest backlog size is 4 packets for sequence $seq(1, 2, 1)$.

5.3.5 Computation of the sequences at the input ports of switch $S1$

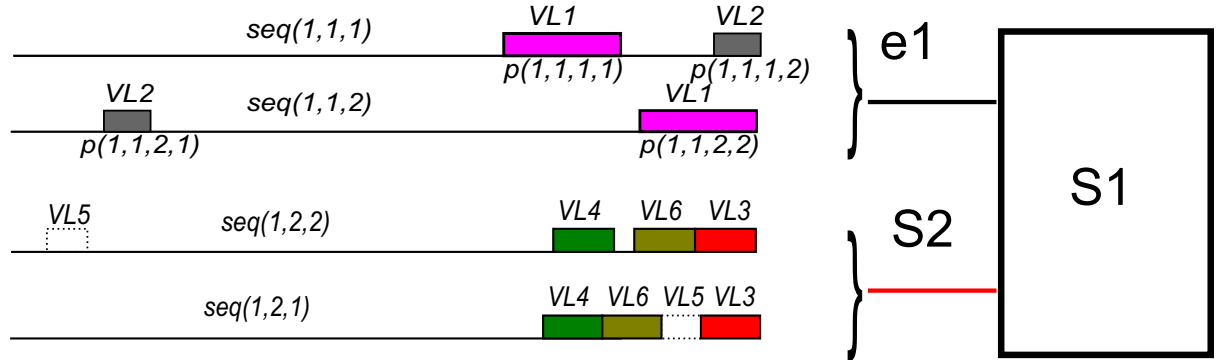


Figure 5.7 – Sequences generated at input of Switch $S1$.

The next hop in the path of VL $v3$ is switch $S1$ output port. At this port VL $v1$, $v2$, $v3$, $v4$ and $v6$ compete for link $link(3, 1)$. For link $link(1, 1)$, we have two sequences, $seq(1, 1, 1)$ and $seq(1, 1, 2)$ for VLs $v1$ and $v2$. For link $link(1, 2)$, we have two sequences, $seq(1, 2, 1)$ and $seq(1, 2, 2)$ computed in previous step. Also at this port VL $v5$ is not present so we need to remove packets of VL $v5$ from sequences $seq(1, 2, 1)$ and $seq(1, 2, 2)$. These sequences are shown in figure 5.7. The link $link(1, 2)$ has total of two sequences: $seq(1, 2, 1)$ and $seq(1, 2, 2)$. So at this output port total combinations are $2 * 2 = 4$. If we were considering all sequences in previous step, there would be $2 * 4 = 8$ sequences at the input of switch $S1$.

5.3.6 Computation of the sequences at the output of switch S_1

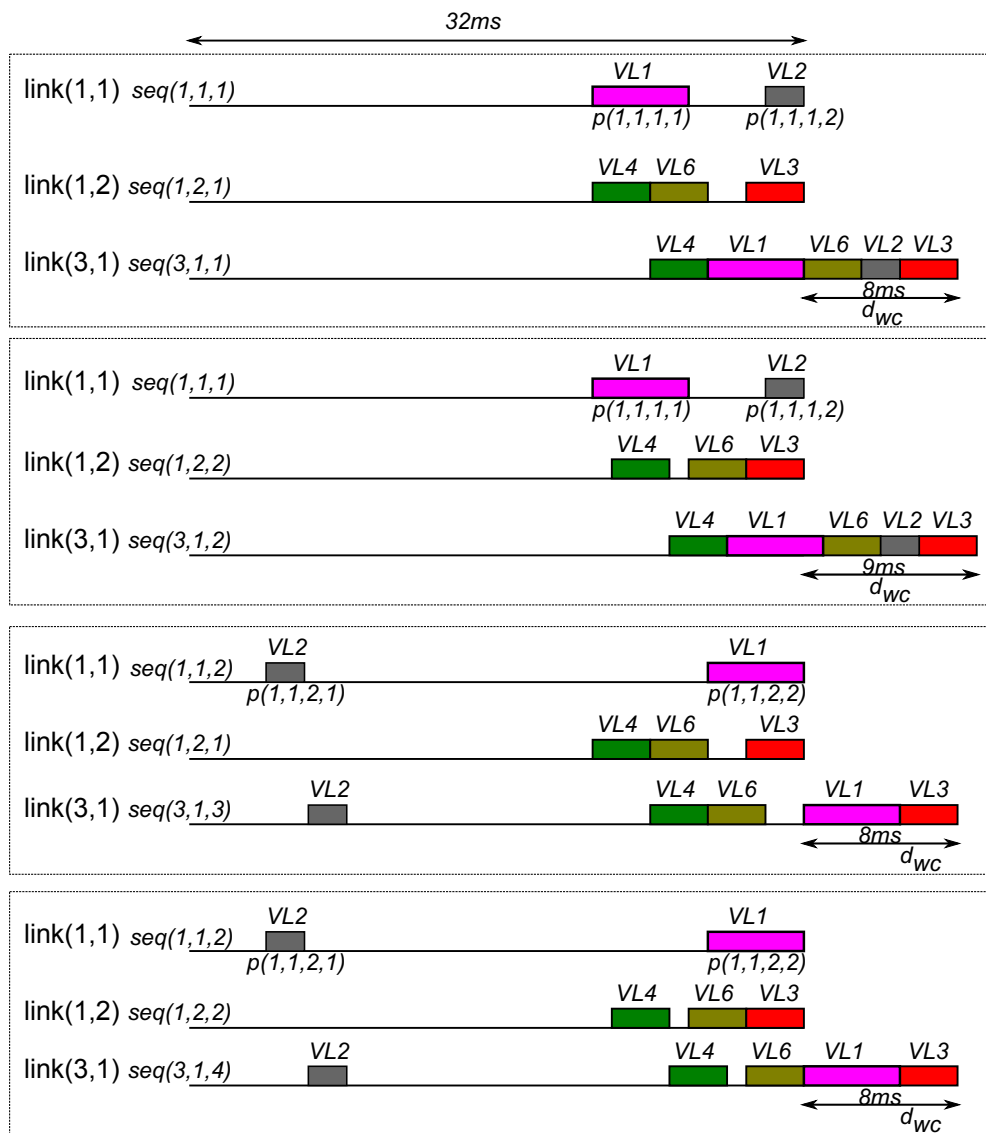


Figure 5.8 – Construction of sequences at output of Switch S_1 Port 1.

Figure 5.8 illustrates the sequences computed at the output port of switch S_1 , by first considering packet $p(1,1,1,2)$ with $seq(1,2,1)$ and $seq(1,2,2)$ and then packet $p(1,1,2,2)$ with $seq(1,2,1)$ and $seq(1,2,2)$. The output of the local algorithm at this port is set of 4 sequences from $seq(3,1,1)$ to $seq(3,1,4)$ for link $link(3,1)$. The local worst case delay at this port for VL v3 is 9ms. One important fact to note is that the local worst case delay at first output port in path of v3 i.e at switch S_2 does not lead to worst case at switch S_1 . In this case it's the second scenario of switch S_2 that leads to worst case at switch S_1 , that's why we need to check all possible combinations.

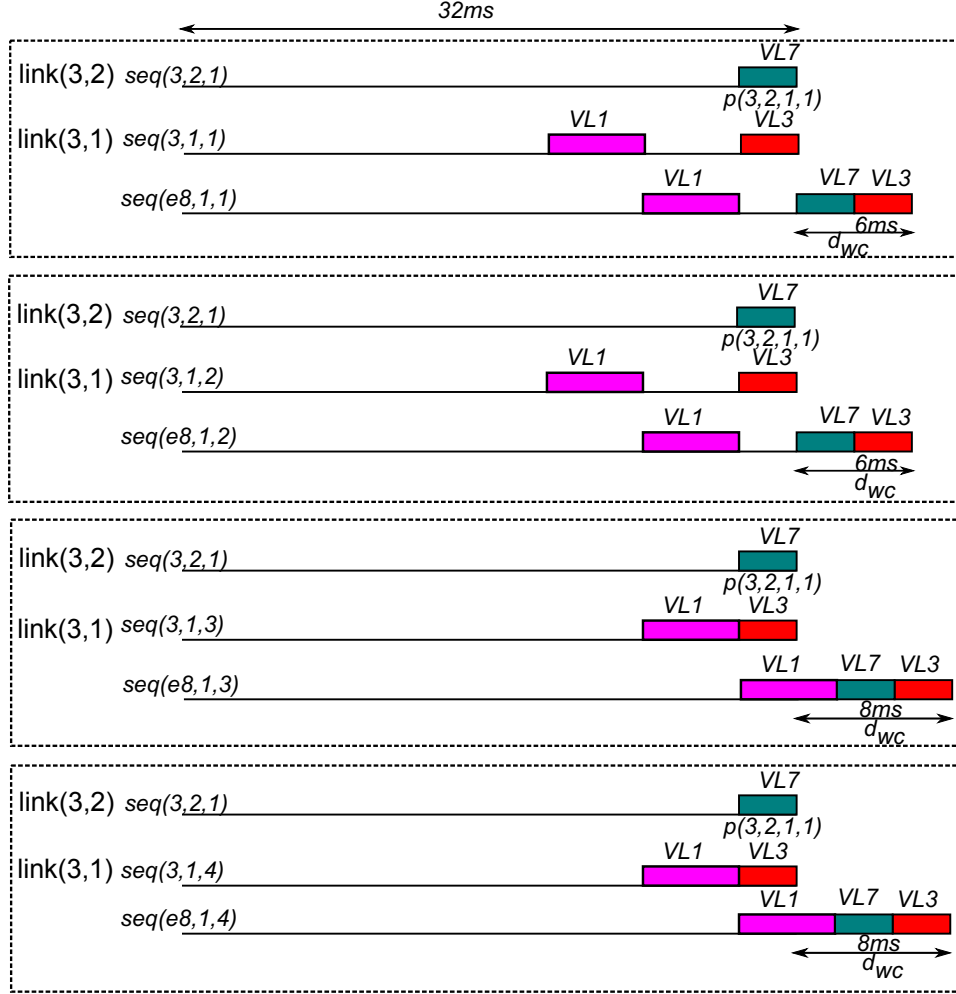


Figure 5.9 – Construction of sequences at output of Switch $S3$ Port 1.

5.3.7 Computation of the sequences at the input ports of switch $S3$

Next and last output port in path of VL v3 is switch $S3$ output port. At this port there is a set of four sequences from link $link(3,1)$ and VL v7 from end system e6 also joins this output port from link $link(3,2)$. So first we compute sequence from link $link(3,2)$. There is only one sequence from this link: $seq(3,2,1)$ consisting of VL v7 packet $p(3,2,1,1)$. Moreover, VLs v2,v4 and v6 are not present at this link so we remove their packets from set of sequences on link $link(3,1)$.

5.3.8 Computation of the sequences at the output of switch S_3

There is only one packet in sequence $seq(3, 2, 1)$ hence total sequences at output of switch S_3 are $1 * 4 = 4$. Figure 5.9 illustrates these combinations by considering packet $p(3, 2, 1, 1)$ with each sequence from link $link(3, 1)$. The end-to-end delay for VL v3 is summarized in table 5.2. The worst case delay is $26ms$ and corresponds to the scenario highlighted in bold. The VLs which does not cross or share the path of the VL under study, have no effect on the computation complexity for a given VL. In this example, VL v8 does not share path with reference VL v3 and hence it is not included in computation for VL v3.

e2(ms)	S2P1(ms)	S1P1(ms)	S3P1(ms)	Total(ms)
3	seq(1,2,1)=7	seq(3,1,1)=8	seq(e8,1,1)=6	24
		seq(3,1,3)=8	seq(e8,1,3)=8	26
	seq(1,2,2)=6	seq(3,1,2)=9	seq(e8,1,2)=6	24
		seq(3,1,4)=8	seq(e8,1,4)=8	25

Table 5.2 – End to End worst case delay for VL v3

5.4 Evaluation of the sequence based approach

The algorithm proposed in the previous section has been applied on the medium size AFDX network shown in figure 5.12 which includes 18 end systems and 58 VLs. The number of VLs involved in the worst-case delay analysis depends on the VL under study. The algorithm computes the exact worst-case end-to-end delay in less than 1 hour for the VLs of the network in figure 5.12 as long as at most 50 VLs are involved in the computation. This is illustrated in table 5.4 (column *Using Sequences*). As a comparison, the model checking approach presented in [Charara 2006a] does not finish with more than 8 VLs. As compared to timed automata based approach presented in Chapter 4, we can almost double the size of the network for which exact end to end communication delays can be determined. Therefore, the algorithm proposed in this chapter increases the size of the configurations which can be analyzed, thanks to a drastic reduction of the search space. Although the underlying principle and properties are same in both approaches, the results are better for the Java based tool because it is designed specifically for this purpose and handles state space better than timed automata based approach using UPPAAL software.

When more than 50 VLs are involved in the worst-case delay analysis, the algorithm does not finish execution in a reasonable time. Then, the idea is to stop the execution after a

predetermined duration, e.g. 1 hour and use the results computed in this duration. At this instant, a subset of the scenarios which are candidate for the worst-case has been tested. The result of the computation is the highest end-to-end delay obtained with this subset of scenarios. Obviously, the scenarios leading to the worst-case end-to-end delay can be out of the tested subset. Consequently, the result of the computation can be smaller than the worst-case end-to-end delay. Then, a heuristic is applied in order to test first the scenarios which are more promising for the worst-case delay. This is obtained by sorting the generated sequences at the first output port by decreasing order of worst case delays in this port. The worst case delay sequence at the first output port is propagated to the next output port where the same process is repeated. It finishes at the end of the path. Then, the procedure is repeated for second sequence at the first output port, and so on. This approach is explained with the example shown in figure 5.10.

Let's suppose VL $v1$ has two packets $p(1, 1, 1, 1)$ and $p(1, 1, 1, 2)$ in its periodic sequence and starts at end system $e1$ and its destination is end system $e14$ following the path $e1-S_1-S_2-S_3-e14$. At output port P_1 of switch S_1 , VL $v1$ from end system $e1$ competes for the access of link $link(2, 1)$ with VL $v4$ from end system $e2$. Let's suppose VL $v4$ has two packets $p(1, 2, 1, 1)$ and $p(1, 2, 1, 2)$ in its periodic sequence. Depending upon the number of packets in input links of the first switch, all the output sequences of the first switch are calculated. In this case VL $v1$ and VL $v4$ have two packets each so total of 4 sequences ($seq(2, 1, l)$, $1 \leq l \leq 4$) are calculated at output port P_1 of switch S_1 . In next step these sequences are sorted in decreasing order of the waiting time for the frame/packet under study. Then, the first output sequence in the sorted list will be used as input sequence for the subsequent output port in the path of VL $v1$. In this case, let's assume that $seq(2, 1, 2)$ is the first sequence after sorting, then this sequence will be used as input sequence for switch S_2 at link $link(2, 1)$ and competes with VL $v22$ for the access of link $link(3, 1)$ at output port P_1 of switch S_2 . Let's assume that VL $v22$ has two packets $p(2, 2, 1, 1)$ and $p(2, 2, 1, 2)$, so there will be 2 possible sequences, i.e sequence $seq(3, 1, 1)$ and $seq(3, 1, 2)$. We take the first sequence $seq(3, 1, 1)$. The $seq(3, 1, 1)$ is used as input sequence at switch S_3 and competes with VL $v43$ for access of the output port P_1 of switch S_3 . This is the destination port for the VL under study. Let's assume that VL $v43$ has two packets $p(3, 2, 1, 1)$ and $p(3, 2, 1, 2)$, then all possible output sequences at this port will be $seq(e14, 1, 1)$ and $seq(e14, 1, 2)$. The end to end delay is calculated by adding largest waiting times at each output port and is stored in a list. This completes one pass of the algorithm. The process is repeated again for remaining sequences i.e. using $seq(2, 1, 2)$ and $seq(3, 1, 2)$ and so on, as shown in Table 5.3. The algorithm can be interrupted and stopped at any time after the calculation of the first pass, in which case the result computed so far is analyzed and largest value of worst case end to end delay computed so far is chosen as under approximation of the exact end to end worst case delay.

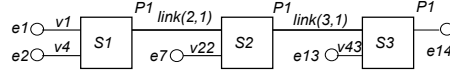


Figure 5.10 – Network for reachable end to end delay illustration.

S1P1	S2P1	S3P1
$seq(2, 1, 2)$ (8ms)	$seq(3, 1, 1)$ (10ms)	$seq(e14, 1, 1)$ (13ms)
		$seq(e14, 1, 2)$ (13ms)
	$seq(3, 1, 2)$ (10ms)	$seq(e14, 1, 1)$ (13ms)
		$seq(e14, 1, 2)$ (13ms)
$seq(2, 1, 1)$ (7ms)	$seq(3, 1, 1)$ (9ms)	$seq(e14, 1, 1)$ (12ms)
		$seq(e14, 1, 2)$ (12ms)
	$seq(3, 1, 2)$ (9ms)	$seq(e14, 1, 1)$ (12ms)
		$seq(e14, 1, 2)$ (12ms)
$seq(2, 1, 3)$ (6ms)	$seq(3, 1, 1)$ (8ms)	$seq(e14, 1, 1)$ (11ms)
		$seq(e14, 1, 2)$ (11ms)
	$seq(3, 1, 2)$ (8ms)	$seq(e14, 1, 1)$ (11ms)
		$seq(e14, 1, 2)$ (11ms)
$seq(2, 1, 4)$ (5ms)	$seq(3, 1, 1)$ (7ms)	$seq(e14, 1, 1)$ (10ms)
		$seq(e14, 1, 2)$ (10ms)
	$seq(3, 1, 2)$ (7ms)	$seq(e14, 1, 1)$ (10ms)
		$seq(e14, 1, 2)$ (10ms)

Table 5.3 – End to End worst case delay under approximation algorithm illustration.

This computation of an exact worst case delay (or an under approximation of this delay) can be used in order to compute the pessimism (or an estimation of this pessimism) of the results obtained by the existing Network Calculus and Trajectory approaches, as illustrated in figure 5.11. Table 5.4 gives this information for the configurations which have been analyzed.

5.5 More Improvements and reduction in scenarios

So far we have only considered strictly periodic VLs in our calculation. But real AFDX network has Sporadic VLs. Difference between strictly periodic and sporadic VLs is that while periodic VLs repeat at strictly constant time intervals, the sporadic VLs have a limit on *minimum*

No of VLs	Model Checking (TA)	Using Sequences	Pessimism in NC
8	70min	<1min	1%
16	-	1min	4%
32	-	19min	6%
64	-	stopped after 1 hr	9%

Table 5.4 – Performance comparison of algorithm.

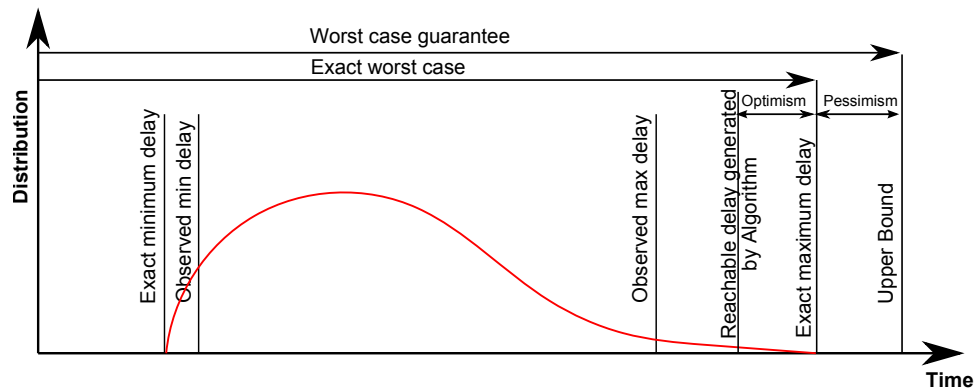


Figure 5.11 – Pessimism of computed upper bounds.

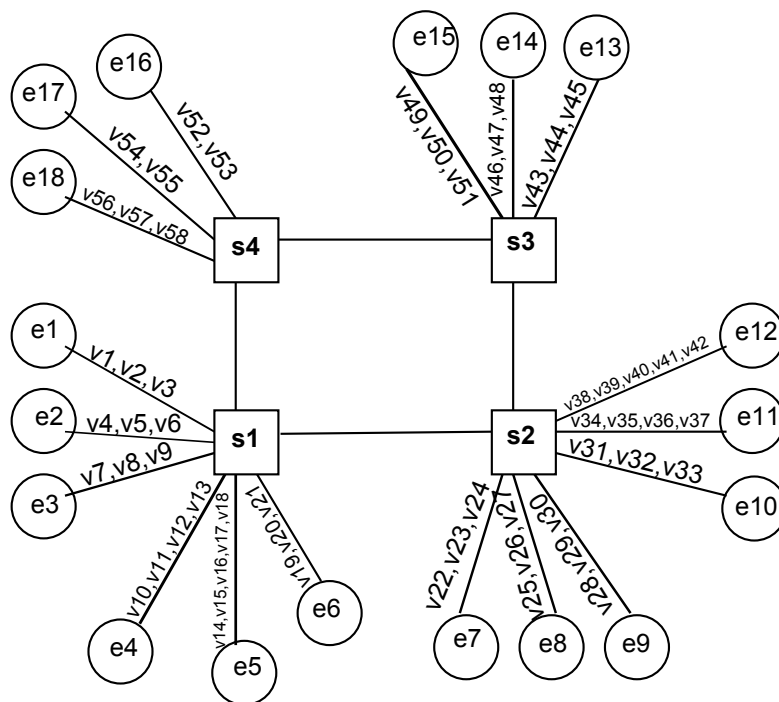


Figure 5.12 – Medium sized AFDX network

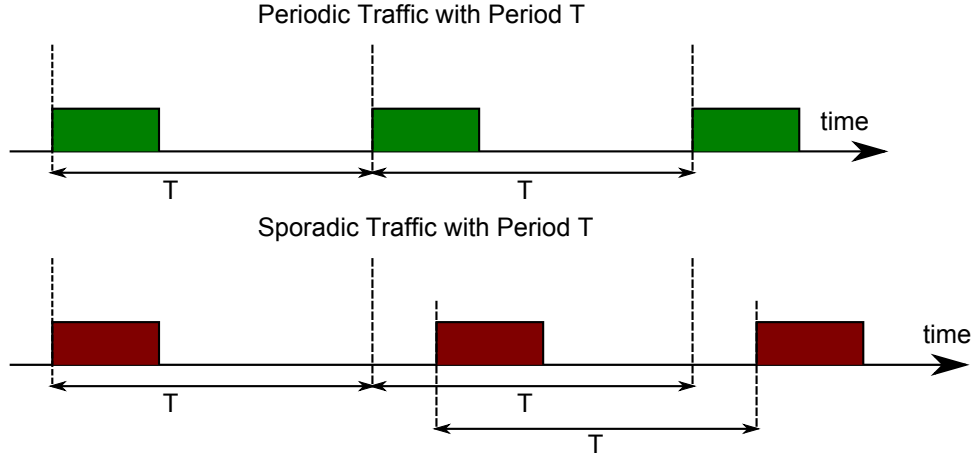


Figure 5.13 – Periodic vs Sporadic traffic

interval between two consecutive occurrences. More precisely, packets of sporadic VL can arrive with gap of any time interval which is greater than or equal to its BAG value while for strictly periodic VL packets can only arrive with gap of time interval strictly equal to its BAG value, as shown in figure 5.13. In order to generate maximum possible data from a sporadic VL, it must strictly transmit a packet at each minimum interval T and hence in worst case scenario sporadic traffic becomes equal to strictly periodic traffic, but with no *offsets*.

5.5.1 Modeling of Sporadic traffic

We have already seen how a strictly periodic traffic data can be modeled in section 5.1.2. When VLs are strictly periodic, we can assign offsets in order to reduce worst case delays and to evenly distribute traffic in the network. But with sporadic VLs, we cannot put constraint of strictly defined offsets between different packets of the VL. Therefore, we model the sporadic VLs with offsets equal to zero *i.e.* packets arrive at the same time and hence we must consider all possible orders of transmissions. This generates huge number of scenarios at each end system (factorial of number of packets in one hyper period of the end system). For example, on an end system with 5 VLs having 128ms BAG (Hyper Period) for each VL, the total number of scenarios at this end system will be $5! = 120$. For BAG values other than 128ms (maximum possible BAG value or Hyper Period), this number becomes even greater because more packets are generated in the hyper period of 128ms. This is illustrated in figure 5.14. In this illustration, there are 13 packets in one hyper period, therefore total scenarios for this end system will be $5! \times 2! \times 4! \times 2! = 11520$. In short, the closed form complexity of such approach will be $\mathcal{O}(n * n!)$.

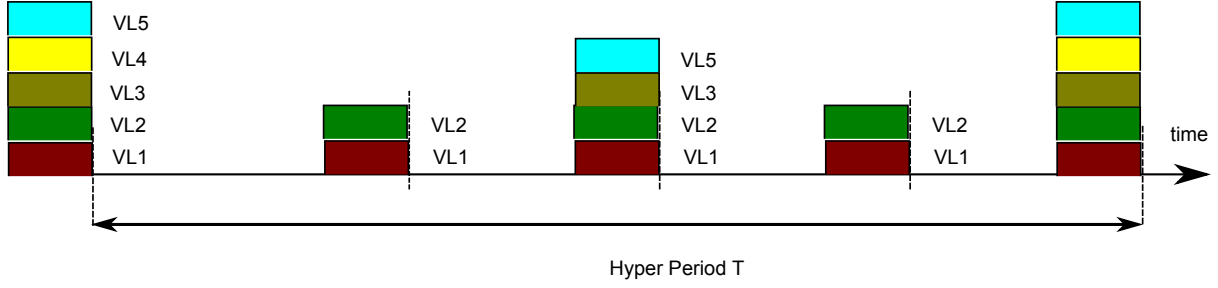


Figure 5.14 – Sporadic traffic in hyper period

5.5.2 Further reduction of scenarios

In this section we establish more properties of the network that will be used to reduce number of scenarios which are candidate for worst case end-to-end delay.

Property 3: Order of packets in Switch output port.

Assuming, data rate of input link and output link is same, if two packets x_1 and x_2 cross a switch output port and there is no existing backlog in the port, then following is true:

- if $x_1 \geq x_2$ then there will be no idle time between x_1 and x_2 at the output of the switch port.
- if $x_1 < x_2$ then there will be idle time between x_1 and x_2 at the output of the switch port. This idle time is equal to $x_2 - x_1$.

Proof: The proof is very simple and intuitive. AFDX switch works on store and forward mechanism and its output link is a FIFO queue. A smaller packet is transmitted before the larger packet has fully been received which results in idle time between these packets as shown in figure 5.15. ■

Property 4: Idle time and queuing delays. In an AFDX switch, an idle time on input links can not increase queuing delay at the given FIFO queue of output link.

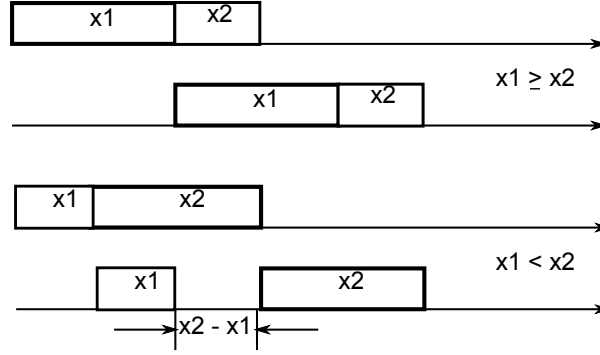


Figure 5.15 – Order and size of packet in switch output port

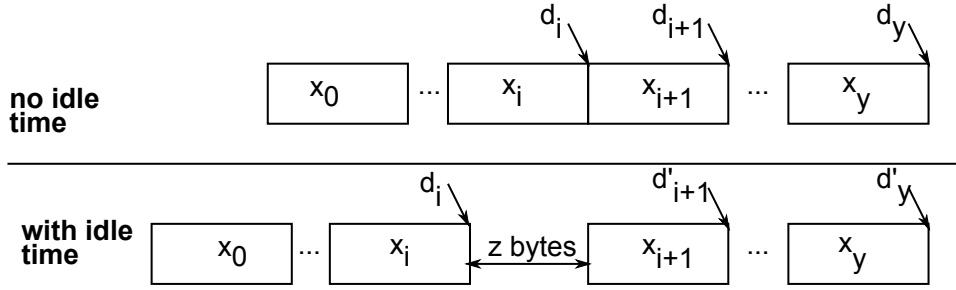


Figure 5.16 – Idle time and queuing delay at a switch output port.

Proof: If data rate on input links of an AFDX switch is same as data rate in output links, then the queuing delay of a packet in its input links competing for a given output link, is directly proportional to the amount of data waiting for the transmission in the output port queue. If more packets are arriving at the output queue than the packets being transmitted, then the data in queue will start to increase. On the other hand, if no packet is being received, *i.e* there is idle time on the input links, then data in output port queue will reduce. This implies that having idle time in input links will reduce the amount of data being stored in the queue and hence will result in lesser queuing delay at the output port.

Consider a sequence of frames x_0, \dots, x_y coming from a single input link and transmitted on a single output link. Let us assume that there is no idle time within the input sequence. Let us assume, queuing delay at the output port at the reception of last frame x_y is d_y bytes, as shown in upper part of figure 5.16. Now, let us consider that an idle time of z bytes is inserted within the input sequence, between frames x_i and x_{i+1} with $0 \leq i < y$, as shown in lower part of figure 5.16. The queuing delay at the output port at the reception of last frame x_y in this case is denoted by d'_y bytes. Without loss of generality, we can assume that the last packet x_y

is received at the same time instance. If d_i denotes the queuing delay in bytes at the reception of frame x_i , then d_i is same for both cases because both cases are identical till frame x_i , as shown in figure 5.16. At the reception of frame x_{i+1} , the queuing delay in bytes for the first case with no idle time is given by: existing data in queue + received data size - the amount of data transmitted during this period. If $s(x_i)$ denotes the size of packet x_i , then we can say that:

$$\begin{aligned} d_{i+1} &= \max(d_i - s(x_{i+1}), 0) + s(x_{i+1}) \\ &= \max(d_i, s(x_{i+1})) \end{aligned}$$

For the second case with idle time of z bytes, the queuing delay in bytes at the reception of frame x_{i+1} is given by:

$$\begin{aligned} d'_{i+1} &= \max(d_i - z - s(x_{i+1}), 0) + s(x_{i+1}) \\ &= \max(d_i - z, s(x_{i+1})) \end{aligned}$$

Consequently, comparing the both cases, we have:

$$d'_{i+1} \leq d_{i+1}$$

Taking into account the fact that sequence of frames in both cases are identical after the frame x_{i+1} , we can conclude that:

$$d'_y \leq d_y$$

Thus, we can say that adding idle time in the sequence of received frames can never increase the queuing delay at a given output port. ■

An interesting result of this property is the fact that if no VL leaves the path of VL under study after joining it and packets are in order of decreasing packet sizes (otherwise it will cause an idle time on the output link according to Property 3) then this will be the only worst case scenario.

In an AFDX network, it is very rare that no VL leaves the path of VL under study. VLs join the VL under study, share part of the path and then leave. This implies that there are more than one cases for worst case end to end delay. We are interested in amount of data present in the queue at the instance when the packet under study arrives at this output port. If idle time appears closer to the packet under study, it will result in lesser delay and vice versa. Hence, earlier the idle time in a sequence, more chances of worse end to end delay. But for exact worst

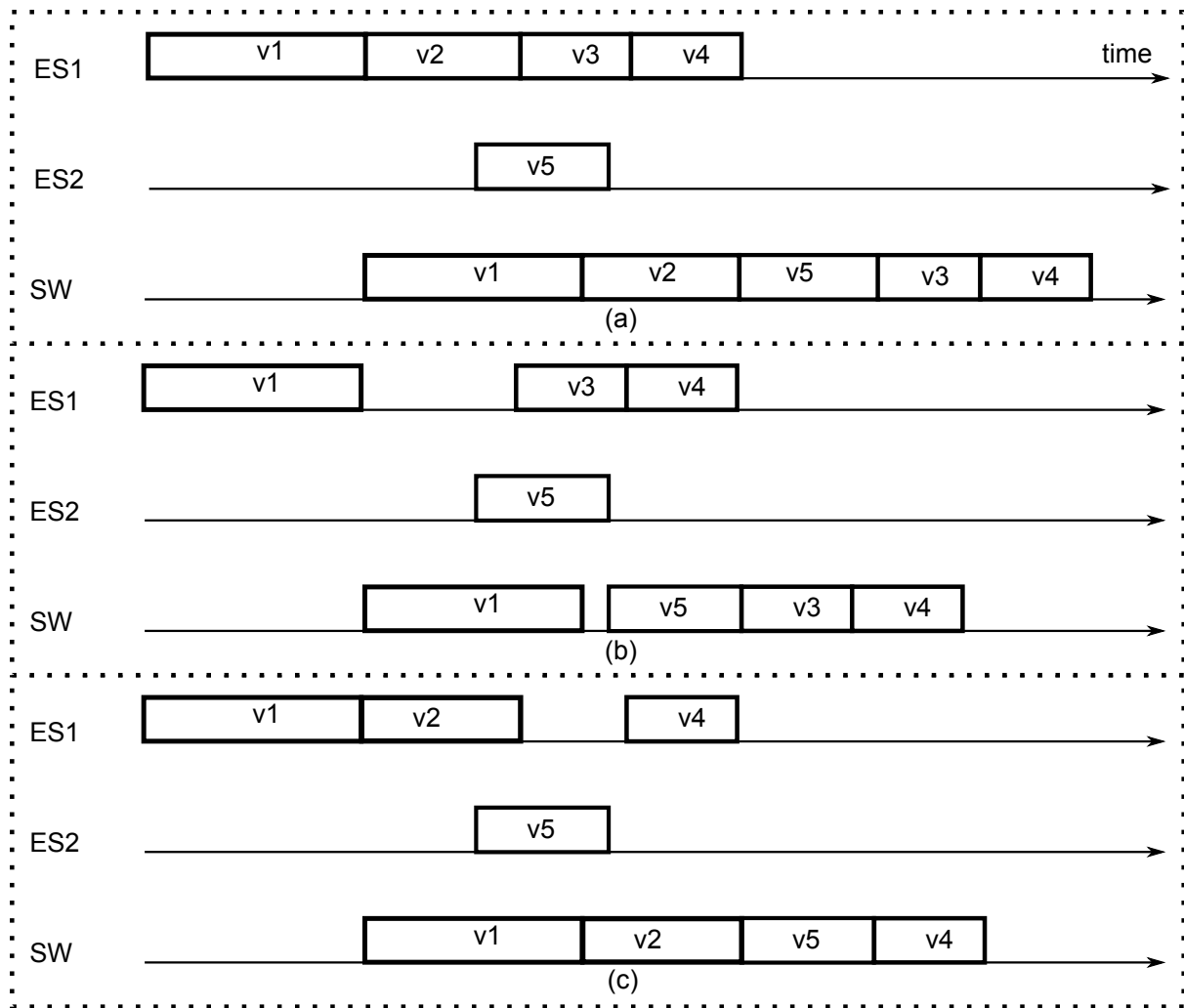


Figure 5.17 – Idle time due to leaving VLs and its impact on packet under study.

case delay, we must check all possible placements of this idle time in the sequence, as shown in figure 5.17, for packet $v5$. In part (a) of the figure 5.17 no VL is leaving and hence packet $v5$ is delayed by packet $v2$. In part (b) of the same figure, packet $v2$ is leaving so it introduces an idle time which results in no delay for packet $v5$; packet $v5$ is immediately transmitted on its arrival because switch output port is not busy. If this idle time was due to packet $v3$ as shown in part (c) of the figure 5.17, then again packet $v5$ will be delayed by packet $v2$ as in case (a). Hence placement of idle time on the link affects the waiting time for the packets on the link. ■

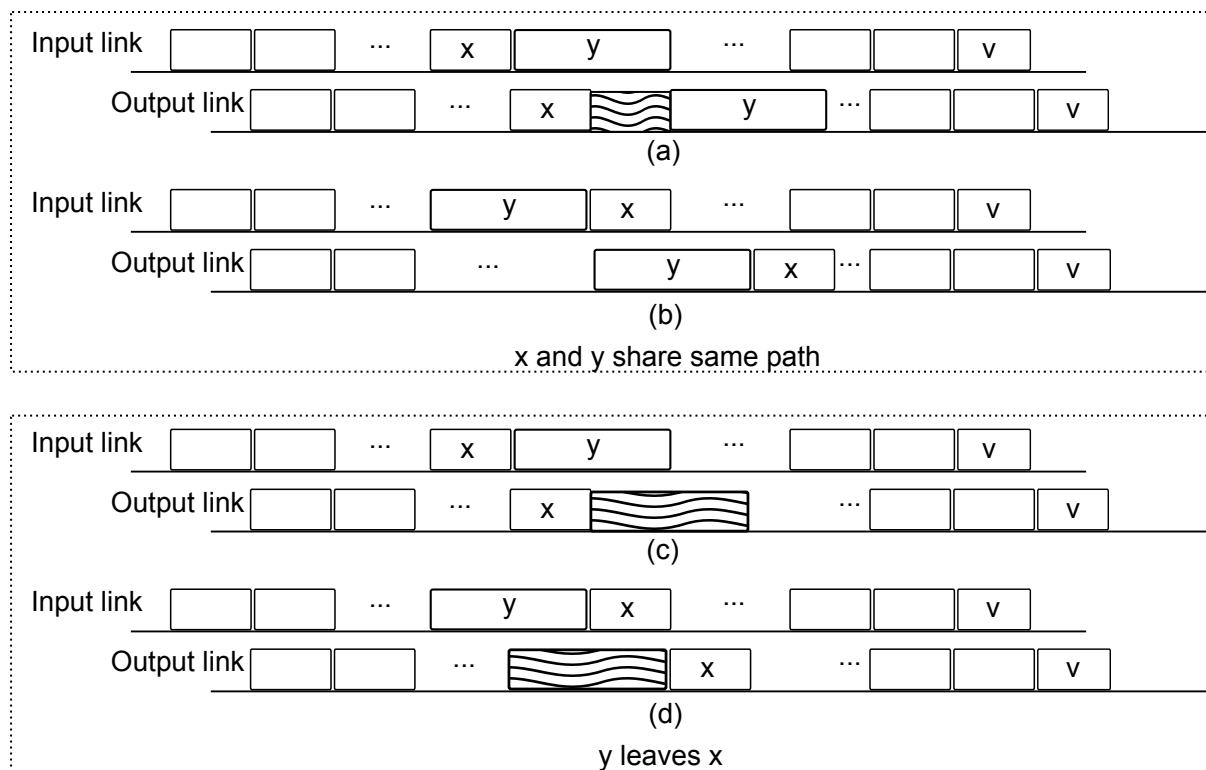


Figure 5.18 – Idle time and queuing delay at a switch output port.

5.5.3 Candidate scenario for worst case delays

Consider a sequence of frames in an input link of the switch as shown in figure 5.18(a). If in figure :

- x shares more nodes with the frame under study (v) than y , and
- $x < y$.

Then we will show that by changing the position of x with y as shown in figure 5.18(b), it can never result in smaller end to end delay for the frame under study. This will allow us to discard all the scenarios where above condition holds true. To prove above condition, we must consider two cases.

- Nodes (switch ports) where x and y are together, and
- the node where y leaves x .

Nodes where x and y are together In this case $y > x$, this implies no idle time in the output link, according to the Property 3 as shown in figure 5.18(b). No idle time also implies that on all such nodes the queuing delay will be worse or equal to original scenario of figure 5.18(a), according to the Property 4.

Nodes where y leaves x In this case y shares less nodes with frame under study than x . When y leaves, there will be idle time in the output link, as shown in figure 5.18(d) but this idle time will be earlier as compared to the original scenario shown in figure 5.18(c). Hence the delay will be worse or equal to original scenario of figure 5.18(c), according to the Property 4.

This condition allows us to reduce number of scenarios which can be candidate for worst case end to end delay.

5.5.4 Algorithm to further reduce number of cases

In order to take advantage of the properties discussed earlier and to reduce the total number of scenarios candidate for the worst case end-to-end delay, we have developed an algorithm which generates a list of scenarios that can be a candidate for the worst case end-to-end delay and eliminates all those scenarios which can not be a candidate for the worst case end-to-end delay

by exploiting the condition expressed in section 5.5.3. The algorithm is shown in Algorithm 2. The basic principal is to make list of scenarios which don't fulfill conditions expressed in section 5.5.3. We start with sorting of VLs according to the number of shared nodes with the frame of VL under study. Then we group these VLs into lists where each VL in a given list shares same number of nodes with VL under study. These lists are then sorted by decreasing packet size of the VLs. We start from the list which has least number of shared nodes, and take each VL from other lists, in sequence of increasing number of shared nodes. By comparing all the VL packet sizes we decide if a particular sequence will satisfy the condition presented in section 5.5.3 or not. We generate only sequences which don't satisfy the condition. As an example to illustrate the algorithm, let us consider 9 VLs named as v_i where $i = 1 \dots 9$ and with packet sizes of 50, 30, 10, 60, 20, 5, 45, 30, 15 respectively as shown in table 5.5. As a first step, all VLs are arranged by increasing number of shared nodes. Then these VLs are grouped into different lists having same number of shared nodes in each list. In the table 5.5.3, VLs v_1, v_2 and v_3 are in list L_0 , VLs v_4, v_5 and v_6 are in list L_1 and VLs v_7, v_8 and v_9 are in list L_2 . In each list L_i these VLs are sorted by decreasing packet size. So they become as shown below:

$$L_0 = \{v_1, v_2, v_3\}$$

$$L_1 = \{v_4, v_5, v_6\}$$

$$L_2 = \{v_7, v_8, v_9\}$$

We start with list L_0 as a partial sequence Sp_0 . Since there are three lists L_0, L_1 and L_2 , therefore there will be two iterations of *while* loop in the algorithm 2 (after taking out L_0 as Sp_0 we are left with only L_1 and L_2). In first iteration we take L_1 . In L_1 we iterate over each VL starting from the last, and compare it with VLs of Sp_0 (which is same as L_0) also starting from last. So, VL v_6 is compared with VL v_3 , and since size of v_6 (5) is less than that of v_3 (10), it can't be added to Sp_0 , same is true for VLs v_2 and v_1 . So loop breaks by adding v_6 to the end of Sp_0 , storing it in $S^!$ and replacing S by $S^!$. For the next VL in L_1 , i.e v_5 , now we have Sp_0 as $\{v_1, v_2, v_3, v_6\}$. For this iteration, v_5 is greater than v_6 so its added to Sp_0 before v_6 and stored as $\{v_1, v_2, v_3, v_5, v_6\}$ in $S^!$. For next VL v_4 in Sp_0 , v_4 is greater than v_3 so its added to Sp_0 before v_3 and stored as $\{v_1, v_2, v_4, v_5, v_3, v_6\}$ in $S^!$. For next VL v_2 in Sp_0 , v_2 is smaller than v_4 . So, loop breaks, we replace S by $S^!$. For the next VL in L_1 , i.e v_4 , now we have following two partial sequences in S :

$$Sp_0 = \{v_1, v_2, v_3, v_5, v_6\}$$

$$Sp_1 = \{v_1, v_2, v_4, v_5, v_3, v_6\}$$

Comparing VL v_4 with VLs of Sp_0 we obtain following partial sequences which are added to $S^!$:

$$S^! = \{\{v_1, v_2, v_3, v_4, v_5, v_6\}, \\ \{v_1, v_2, v_4, v_3, v_5, v_6\}, \\ \{v_1, v_4, v_2, v_3, v_5, v_6\}, \\ \{v_4, v_1, v_2, v_3, v_5, v_6\}\}$$

Comparing VL v_4 with VLs of Sp_1 we obtain following partial sequences which are added to $S^!$:

$$\{v_1, v_2, v_4, v_5, v_3, v_6\}, \\ \{v_1, v_4, v_2, v_5, v_3, v_6\}, \\ \{v_4, v_1, v_2, v_5, v_3, v_6\}$$

At this point we have iterated over all VLs of list L_1 and the set S contains 7 partial sequences, *i.e.*:

$$S = \{\{v_1, v_2, v_3, v_4, v_5, v_6\}, \\ \{v_1, v_2, v_4, v_3, v_5, v_6\}, \\ \{v_1, v_4, v_2, v_3, v_5, v_6\}, \\ \{v_4, v_1, v_2, v_3, v_5, v_6\}, \\ \{v_1, v_2, v_4, v_5, v_3, v_6\}, \\ \{v_1, v_4, v_2, v_5, v_3, v_6\}, \\ \{v_4, v_1, v_2, v_5, v_3, v_6\}\}$$

The same procedure is repeated for VLs v_7 , v_8 and v_9 of L_2 and in the end S contains all the sequences which are candidate for worst case scenario. These sequences are then used to find worst case end-to-end delays. The reduction ratio is huge for these cases. The total number of cases are $9! = 362880$ but with this algorithm we have 60 cases only. The complexity of this algorithm is $\mathcal{O}(n * n^k)$ in the worst case.

These algorithms and concepts have been implemented in the tool developed to calculate the end to end delays for AFDX network and will be used in case study presented in Chapter 6.

VL	Packet Size	No of Shared Nodes	Group
1	50	1	L_0
2	30	1	L_0
3	10	1	L_0
4	60	2	L_1
5	20	2	L_1
6	5	2	L_1
7	45	3	L_2
8	30	3	L_2
9	15	3	L_2

Table 5.5 – Configuration example for Algorithm 2.

Algorithm 2 Algorithm to find candidate scenarios

```

1:  $S = \text{empty}$  is list of possible scenarios.
2:  $S^! = \text{empty}$  is temporary list of possible scenarios.
3: sort all VLs by order of increasing number of shared nodes
4: store VLs  $V_i$  sharing same number of nodes in a separate list  $L_i$ , and sort them by decreasing
   packet size
5: Store all  $L_i$  in  $L$  in order of increasing number of shared nodes
6: if  $\text{sizeof}(L) == 1$  then
7:    $S = L_0$ 
8: else
9:   Take  $L_0$  as a partial sequence  $Sp_0$ 
10:  Add  $Sp_0$  to  $S$ 
11:  while  $L \text{ hasNext}()$  do
12:    for all  $V_i$  in  $L_i$  starting from last do
13:      for all  $Sp_i$  in  $S$  do
14:        for all  $Vsp_i$  in  $Sp_i$  starting from last do
15:          if  $\text{sizeof}(V_i) \geq \text{sizeof}(Vsp_i)$  then
16:            Add  $V_i$  to  $Sp_i$  and store it in  $S^!$ 
17:          else if all VLs ( $Vsp_i$ ) in  $Sp_i$  has been checked then
18:            Add  $V_i$  to the end of  $Sp_i$  and store it in  $S^!$ 
19:          else if  $V_i$  has been added at least once then
20:            break
21:          end if
22:        end for
23:      end for
24:       $S = S^!$ 
25:       $S^! = \text{empty}$ 
26:    end for
27:  end while
28: end if

```

5.6 Conclusion

In this chapter, we have presented an improved method to calculate exact worst case delays for AFDX network. This approach is promising since it increases the maximum size of the networks which can be analyzed. Another advantage of this technique is that it can generate reachable (but close to the exact worst case) scenarios for large networks which can not be computed for exact worst case in reasonable time. It will probably be difficult to analyze a real industrial configuration (more than 1000 VLs) for exact worst case delays but still we can use this technique to obtain valuable information on the pessimism of upper bounds obtained by Network Calculus or Trajectory approach.

The proposed approach can be adapted in order to compute the backlog in each switch output port, by calculating the maximum number of adjacent packets in computed sequences at this switch output port. This is important for certification: it is mandatory to guarantee that buffers of switch will never overflow.

This technique is not limited to AFDX network but can cope with any switched Ethernet network, provided flows are strictly periodic. It can also be used for analysis of different scheduling techniques and finding delay bounds in multi-processor networked architecture.

The first contribution of this work is to explain how the number of scenarios which have to be analyzed for an exact worst case delay computation can be drastically reduced.

The second contribution of this work is to propose an algorithm for the worst case delay computation based on this reduced set of scenarios. This algorithm allows the analysis of larger network configuration than existing model checking approaches in [Adnan 2010b, Charara 2006a].

The third contribution of this work is to show how the proposed algorithm can be adapted for industrial configurations to obtain reachable end-to-end delay cases which are close from the exact worst case.

We have developed a tool based on the algorithms and properties presented in this chapter. This tool was used to analyze an real-life industrial configuration of the AFDX network. The details and results of this case study are presented in the next chapter.

Case Study

Contents

6.1 AFDX network system of industrial scale complexity	113
6.1.1 Understanding the complexity of industrial scale AFDX network	113
6.2 Software Architecture	120
6.3 Results of the Case Study	120
6.3.1 Comparison of results with Network Calculus and Trajectory approach . . .	124
6.4 Conclusion	124

In this chapter we will present the case study of an industrial implementation of AFDX network such as used on Airbus A380 aircraft. This AFDX network was analyzed for worst case end to end delays using sequence based approach presented in Chapter 5 using our own developed tool. Before talking about the results of this analysis, we will present the system first in next section.

6.1 AFDX network system of industrial scale complexity

In this section we will present a real life AFDX network of industrial scale complexity such as used on commercial aircraft *e.g.* Airbus A380. Such a large AFDX network consists of about 1000 VLS constituting more than 6000 paths, using 16 AFDX switches and 123 end systems as illustrated in figure 6.1.

6.1.1 Understanding the complexity of industrial scale AFDX network

The complexity of an industrial scale AFDX network, as far as number of possible scenarios is concerned, is mainly because of following two factors:

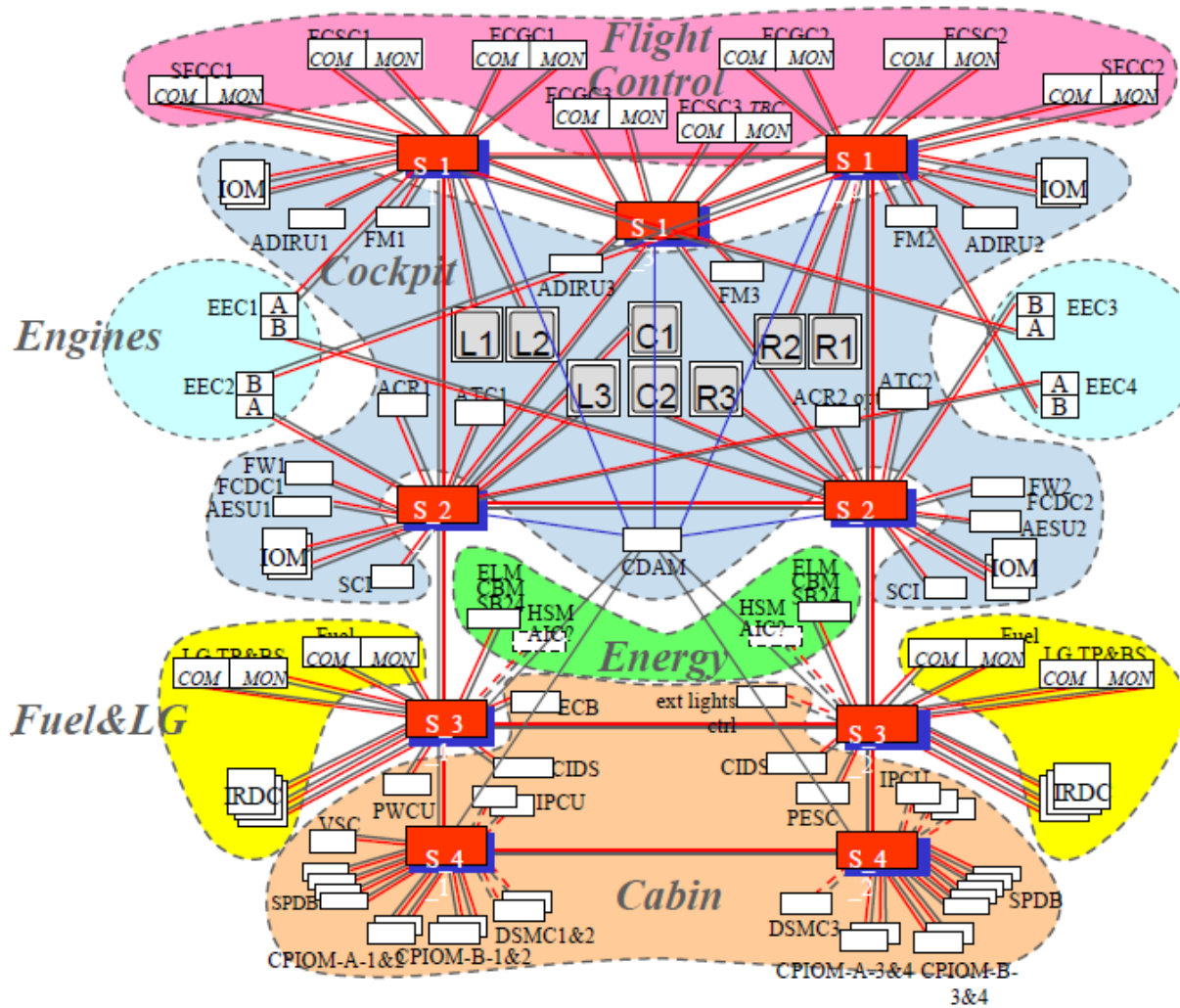


Figure 6.1 – AFDX Network of Airbus A380 aircraft.

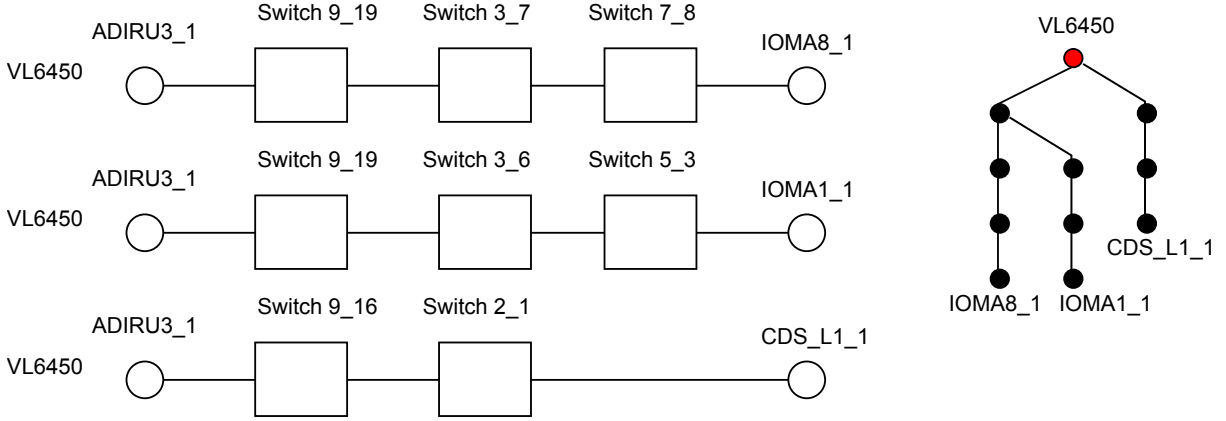


Figure 6.2 – Single VL and its paths with equivalent tree.

Interference of other VL paths with path of VL under study: We have already discussed basics of AFDX network in 2.3. In AFDX network, each VL is a multicast broadcast of data. This is very similar to a *tree* with more than one branch. If we consider a single VL in isolation, it will result in number of paths equal to number of branches of the equivalent tree, as shown in figure 6.2. Each path has different end-to-end delay and is calculated separately. Since there are more than 1000 VLs in the network, it is obvious that many paths or branches interfere with each other by intersecting their paths and/or overlapping each other partially. This means that to calculate end-to-end delay for a single path, we must consider all the VL paths which are directly interfering with path under study and also the paths which are indirectly interfering with path under study. The indirect interference can encompass large number of paths. This is equivalent to *Connected Component* in graph theory concepts, where any two nodes are connected to each other by paths and are not connected to any other nodes of the graph, as shown in figure 6.3 where there are three connected components. Each connected component is equivalent to part of AFDX network that we must take into account for a VL under study because of directly and indirectly interfering VLs. In this case study, number of interfering paths for a single path under study, can be upto 5500 paths out of 6412 total paths.

As an example, let us consider a VL named *VL10151* in this case study. This VL originates from end system *IOMA8_1*, passes through switch SW4 at output port 7 (*AFDX_SW4_7*), then through switch SW7 at output port 19 (*AFDX_SW7_19*) and reaches the destination end system. Considering this VL in isolation leads to a very simple scenario as shown in figure 6.4. Now, If we consider all the VLs that interfere with this VL directly, then the scenario is bit more complex, as shown in figure 6.5 where width of edge indicate number of overlapping paths. In this case we have 297 other VL paths which directly interfere with *VL10151*. And when we include all the directly and indirectly interfering VLs, we have 5449 paths interfering

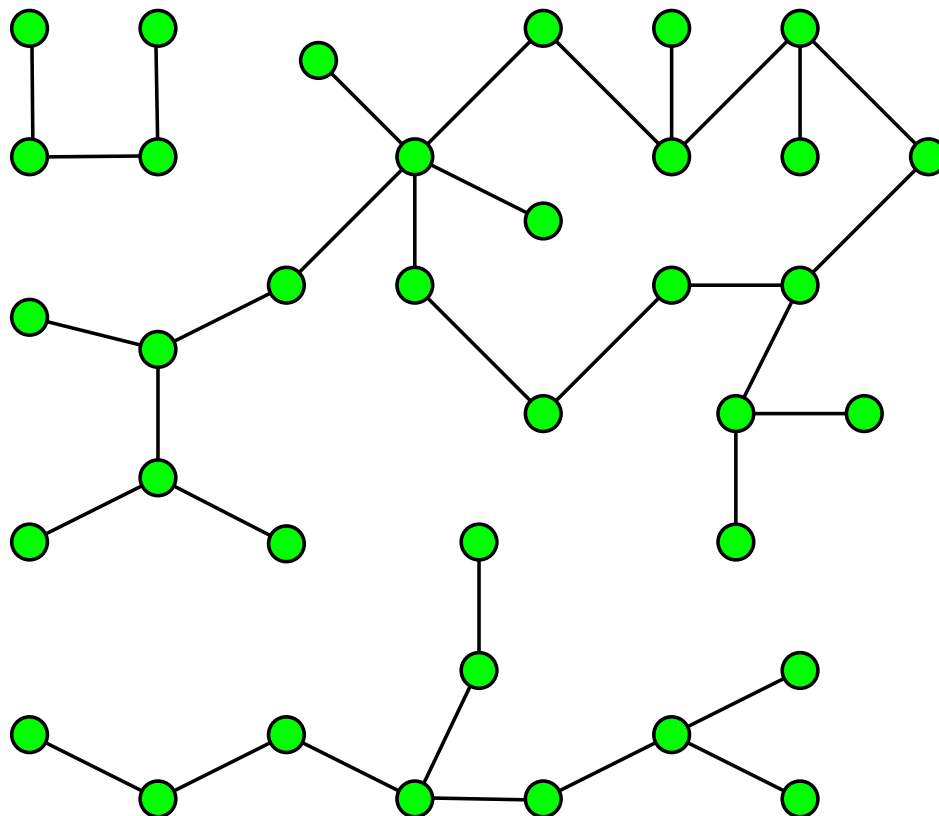


Figure 6.3 – Connected Component of a graph which is equivalent to directly and indirectly interfering VLs.

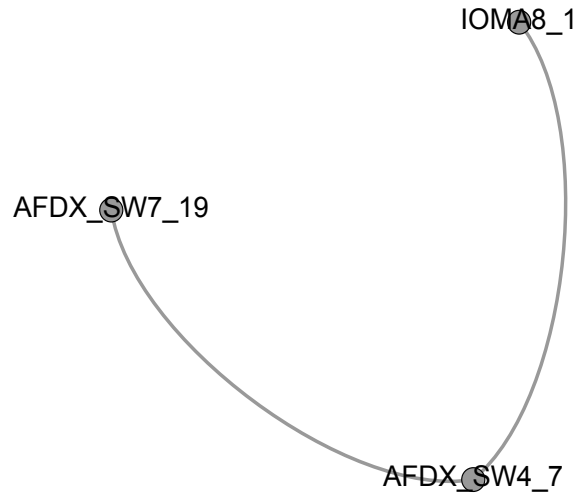


Figure 6.4 – VL10151 in isolation.

with *VL10151* out of 6412 total paths, as shown in figure 6.6 where width of edge indicate number of overlapping paths.

This example of VL *VL10151* demonstrates that the number of cases for even a simple VL path can be huge due to the fact that this path is not in isolation and many other VLs are interfering with it. This makes it even more important to reduce number of scenarios which can be candidate for worst case end to end delay.

Idle times due to leaving VLs: As shown in section 5.1.2, with no idle time between packets, if we arrange packets by decreasing packet size then it will lead us to worst case end-to-end delay provided all packets are going to the same end system *i.e* no VL is leaving the path of VL which is under study. This never happens in industrial AFDX network considered in this case study. When VLs join the VL understudy directly or indirectly, and leave it after sharing a single or multiple switches, they create idle times between packets. The worst case end-to-end delay depends on relative position of this idle time between packets and we must consider all combinations where this idle time can possibly occur. Hence placement of idle time on the link affects the waiting time for the packets on the link, which leads to huge number of possible cases due to large number of paths interfering with each other. The number of cases which can be candidate for worst case end-to-end delays, range from 10^9 to 10^{105} .

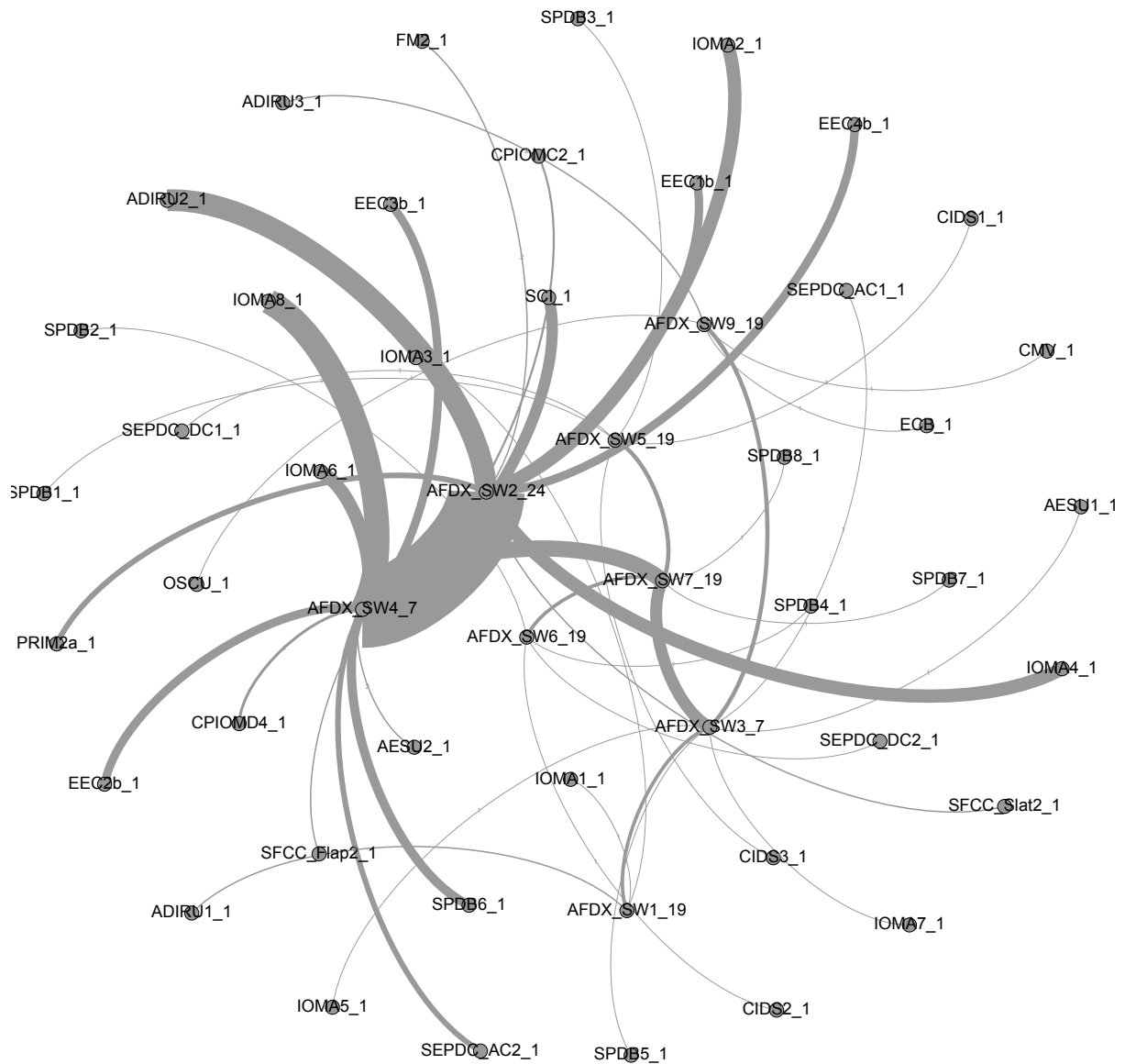


Figure 6.5 – VL10151 directly linked paths.

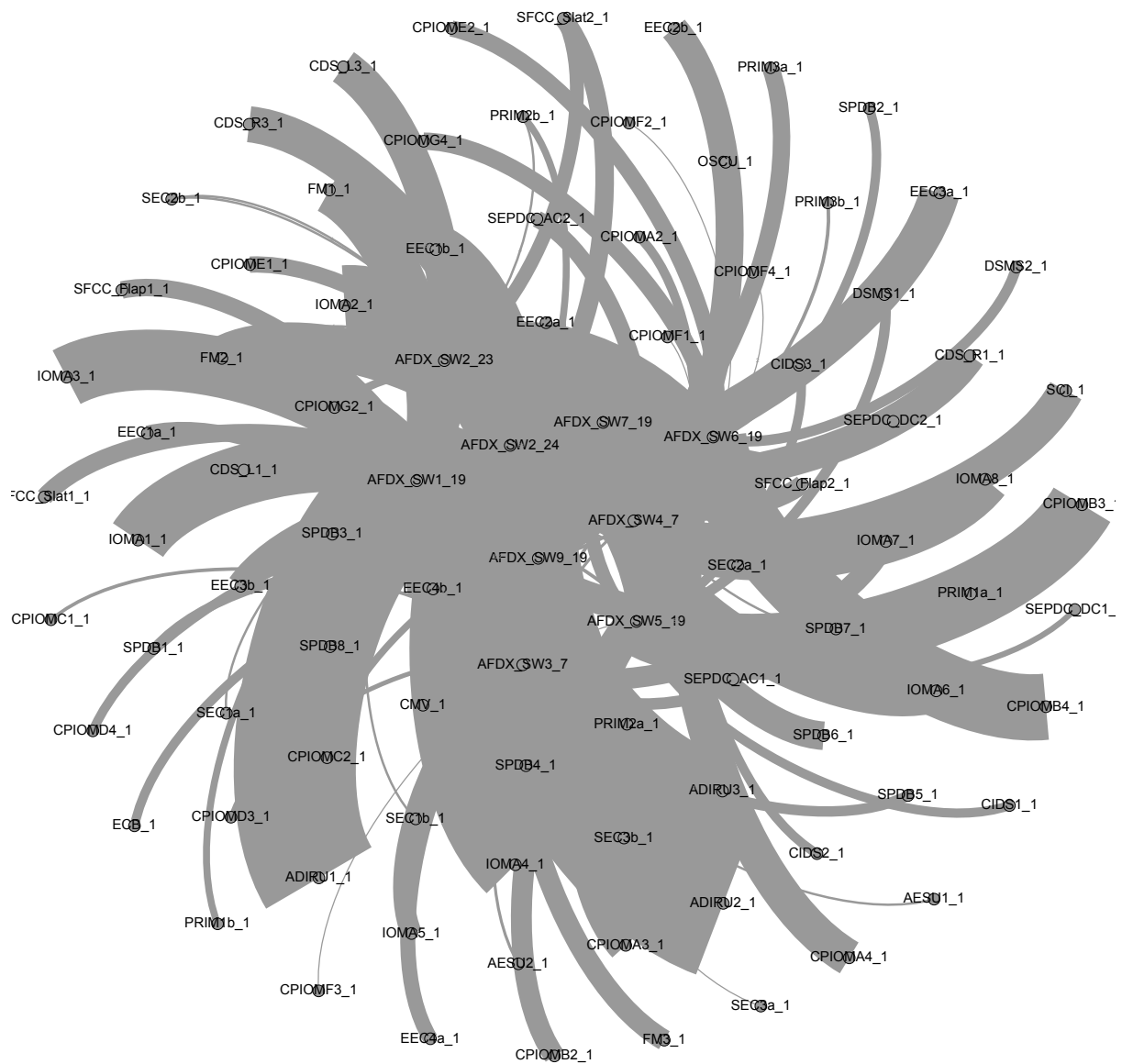


Figure 6.6 – VL10151 all linked paths.

6.2 Software Architecture

The tool for this case study is written in Java. Java was selected for its ability to be portable across multiple operating systems. Java also offers some open source libraries that can be used to develop a distributed application. The tool developed during this research work can be used as stand alone application on a single machine or as a distributed computation application running on multiple machines connected via network. There are two separate versions of the tool which both have same algorithms for computations of end-to-end delays but differ in the way these calculations are managed: in one version computations runs on a single machine while in the other one computations are distributed across multiple machines. More detail about software implementation can be found in Appendix B.

The basic architecture of the software is shown in figure B.1. *Parser* is used to read the AFDX network configuration data and to initialize the internal constructs and variables. *Network Pruning* block builds the essential part of the network that is directly and indirectly linked with the VL under study and removes the *unconnected* part of the whole network. This block is also responsible to generate scenarios which can be candidate for worst case end-to-end delays. In case of distribution computation, *Load Balancer* manages the computation distribution among available machine nodes. In case of single machine, it distributes computations among locally available *cores*. *Compute* module computes the delays and backlogs for a given sequence on a given output port. It also constructs set of resulting output sequences. Finally the *Control* module is responsible for coordination between all modules and for collection of results. The software uses *Flow Based Programming* [Morrison 2010] concepts to implement data flow approach where computations proceed as the data flow from one output port to another. For distributed computations, Java based library named *Java Parallel Processing Framework*(JPPF) is being used. The library can be found at <http://www.jppf.org/>

6.3 Results of the Case Study

The industrial scale network used in this case study interconnects aircraft functions in the avionics domain. It is a large scale network and is composed of two redundant networks (indicated by red and blue colors in figure 6.1). The network is composed of 123 end systems, 8 switches per network with 24 ports on each switch, 984 virtual links and 6412 VL paths (due to multicast characteristics of the VLs). Left part of the table 6.1 shows distribution of VLs according to *BAG* values. The right part of the table 6.1 shows classification of VLs according to the packet size. Table 6.2 shows distribution of VL paths according to number of crossed switches. For an

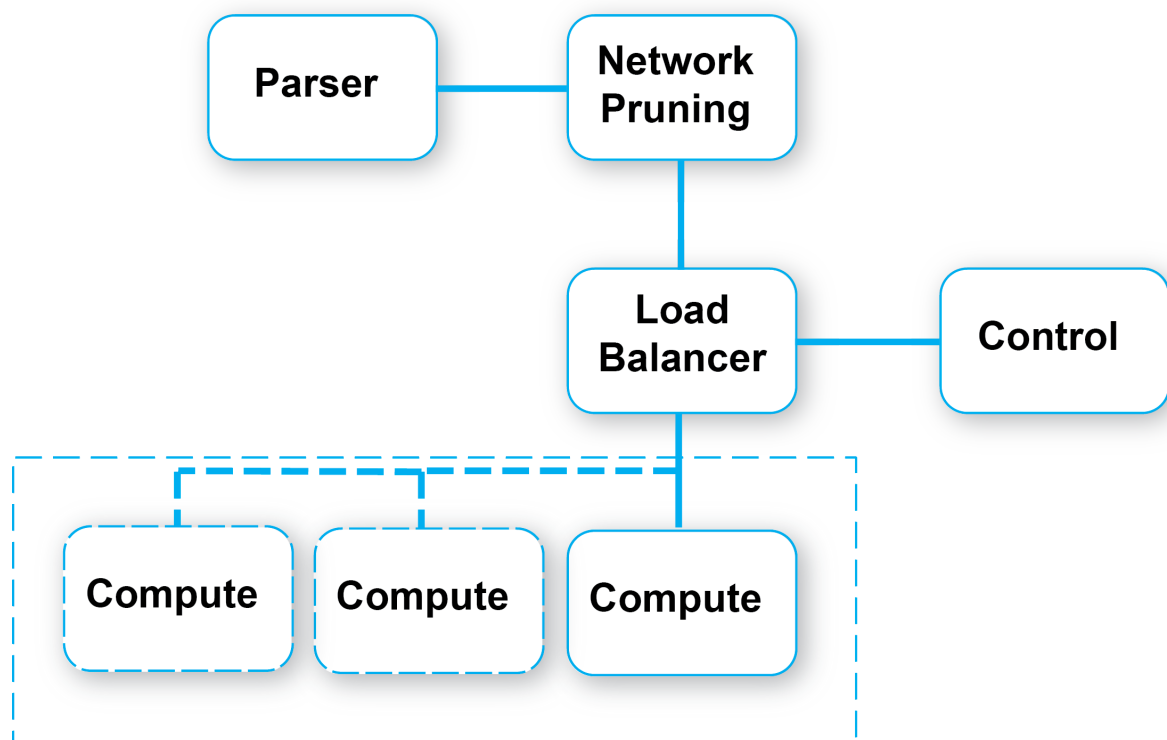


Figure 6.7 – Software architecture.

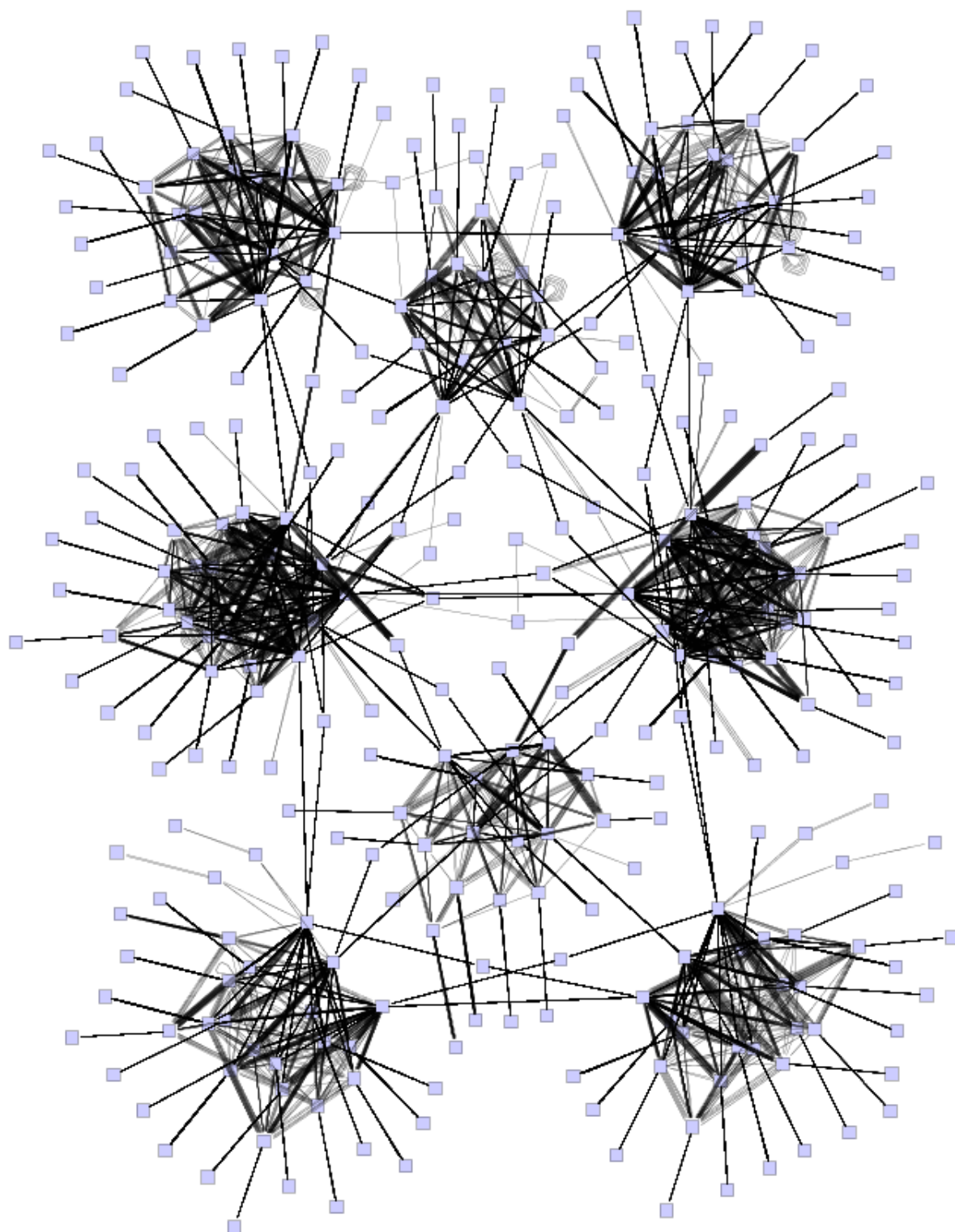


Figure 6.8 – Flows of AFDX network of case study.

BAG (<i>ms</i>)	Number of VLs	Packet size (bytes)	Number of VLs
2	20	0-150	561
4	40	151-300	202
8	78	301-600	114
16	142	601-900	57
32	229	901-1200	12
64	220	1201-1500	35
128	225	> 1500	3

Table 6.1 – AFDX network configuration: BAGs and packet sizes

No of crossed switches	Number of paths
1	1797
2	2787
3	1537
4	291

Table 6.2 – AFDX network configuration: crossed switches number of paths

overview of how these switches and end system are connected with each other, and about the complexity of this communication, figure 6.8 shows actual flows of this network. We can clearly notice how each of these 8 switches are connected to a cluster of end systems.

This industrial AFDX network has links speed of 100 *Mbit/s* on all the links. Overall network load (utilization) is about 10%. Actually, this AFDX network is lightly loaded in order to ensure guarantees on the upper bounds of buffer sizes. There is no global clock in the network and hence packet release times at different end systems are independent.

Let us consider VL *VL10151* as shown earlier in figure 6.4. The maximum packet size for this VL is 567 bytes and the value of *BAG* is 64 *ms*. This VL originates from end system *IOM – A8* port 1. At this end system, there are 6 other VLs competing for the output port of the end system. The worst case queuing delay at this output port is 177.76 μ s. The next output port in the path of this VL is *AFDX_SW4_7* which is AFDX switch number 4 port number 7. At this output port, the queuing delays contributing to worst case end to end delay is 928.72 μ s. The last output port in the path of *VL10151* is *AFDX_SW7_19* which is AFDX switch number 7 port number 19. At this output port the queuing delay is 969.60 μ s. After this output port the VL arrives at the destination end system "FT_SCI". The overall end to end worst case delay is sum of the delays on each output port in the path *i.e* 2076.08 μ s. The Trajectory approach result for the same VL is 177.76, 1000.4, 1106.64 μ s on each output port

respectively with total end to end delay of $2284.8\mu s$. There is about 10% pessimism in the result of Trajectory approach for this VL. If we compare the results on each output port, we notice that the pessimism increases as the number of ports increase in the path of VL. It took more than one hour to analyze all candidate scenarios. But the worst case scenario was reached in just 7 minutes. This is true for most of the VLs. We reach worst case scenario relatively quickly but still we need to analyze all the scenarios which are candidate for the worst case end to end delay.

6.3.1 Comparison of results with Network Calculus and Trajectory approach

We were able to analyze each VL of the network but not all the paths. Some paths were too complex to be analyzed in reasonable time due to a very large number of possible scenarios (in the vicinity of 10^{105}). Out of 6412 paths, we were able to find exact end to end communication delay for 4099 paths. The results of our analysis, *i.e* exact end to end communication delays for 4099 paths, were compared to Network Calculus and Trajectory approach. Figure 6.9 shows the results compared to Trajectory approach and figure 6.10 shows the difference between these results, which is equal to pessimism of Trajectory approach. In some cases, both results are equal which indicates that for those cases trajectory approach finds the exact worst case delay. Figure 6.11 shows the results compared to Network Calculus and figure 6.12 shows the difference between these results, which is equal to pessimism of Network Calculus approach. The actual pessimism in Network Calculus is about 13% on average, and the pessimism in Trajectory approach is about 6% on average.

6.4 Conclusion

Finding exact worst case delays of an industrial scale AFDX network is complex and huge. We were not only able to apply our properties and algorithms on a real industrial scale AFDX network but also we were able to find exact worst case end to end delays for more than 60% of the paths. This shows that model checking or exhaustive simulation can be applied to huge networks. With the exact worst case delays, we can also measure the precise pessimism in upper bounds of Network calculus and Trajectory approach.

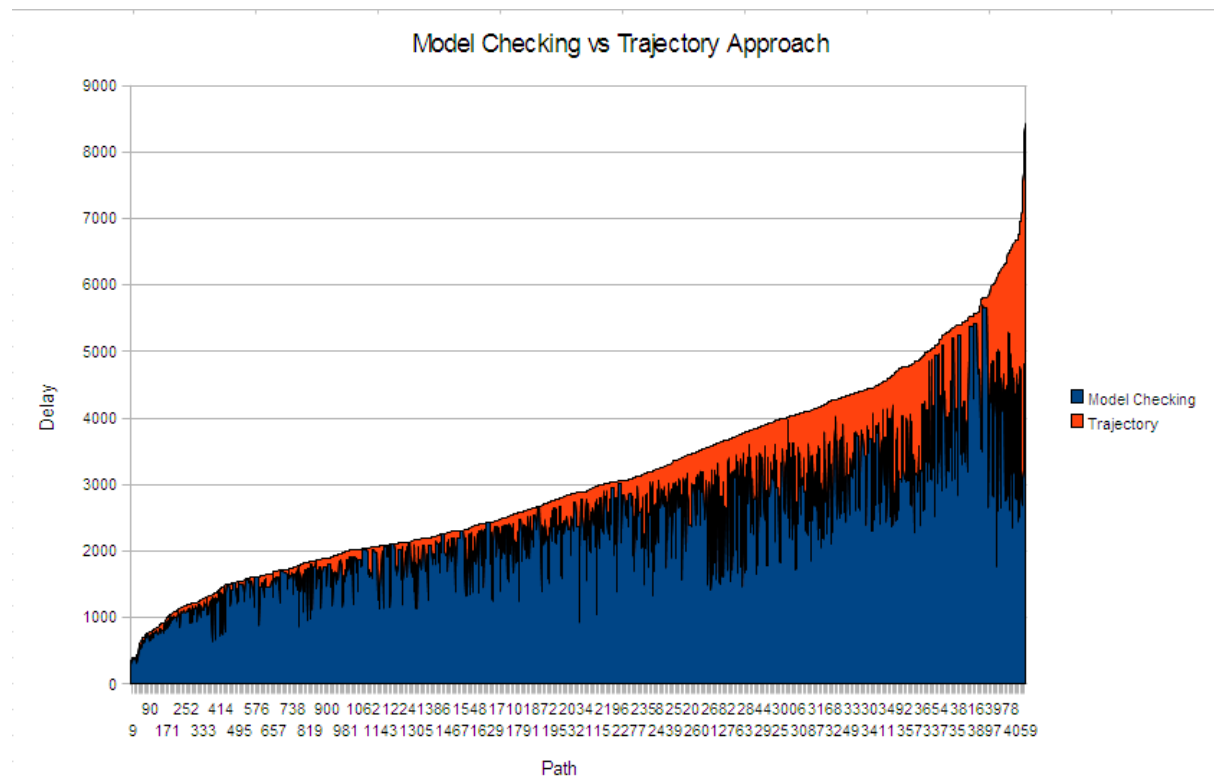


Figure 6.9 – Results of the case study compared to Trajectory approach.

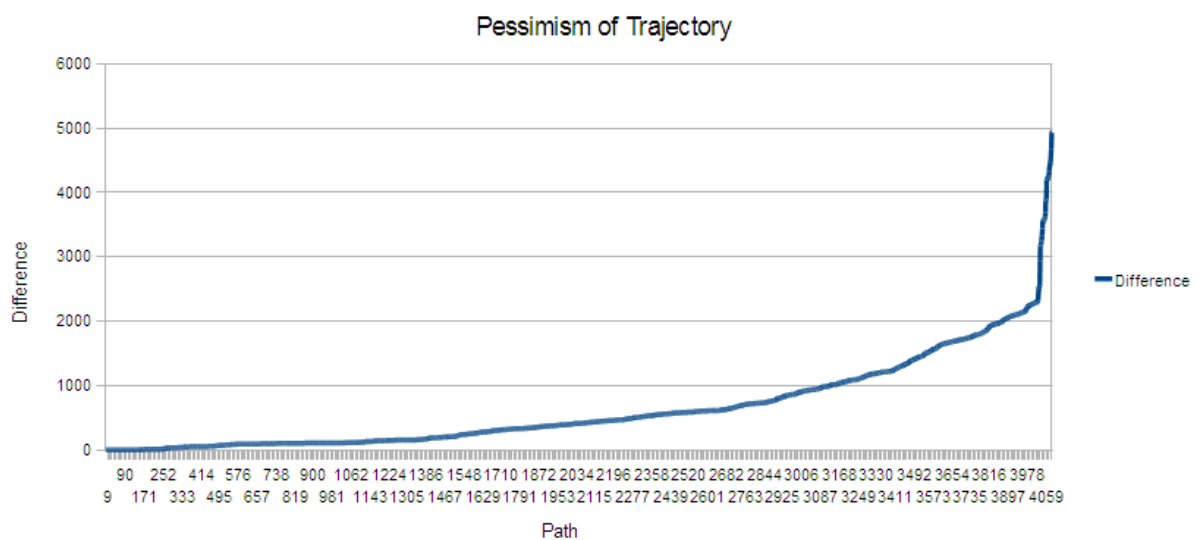


Figure 6.10 – Pessimism in Trajectory approach.

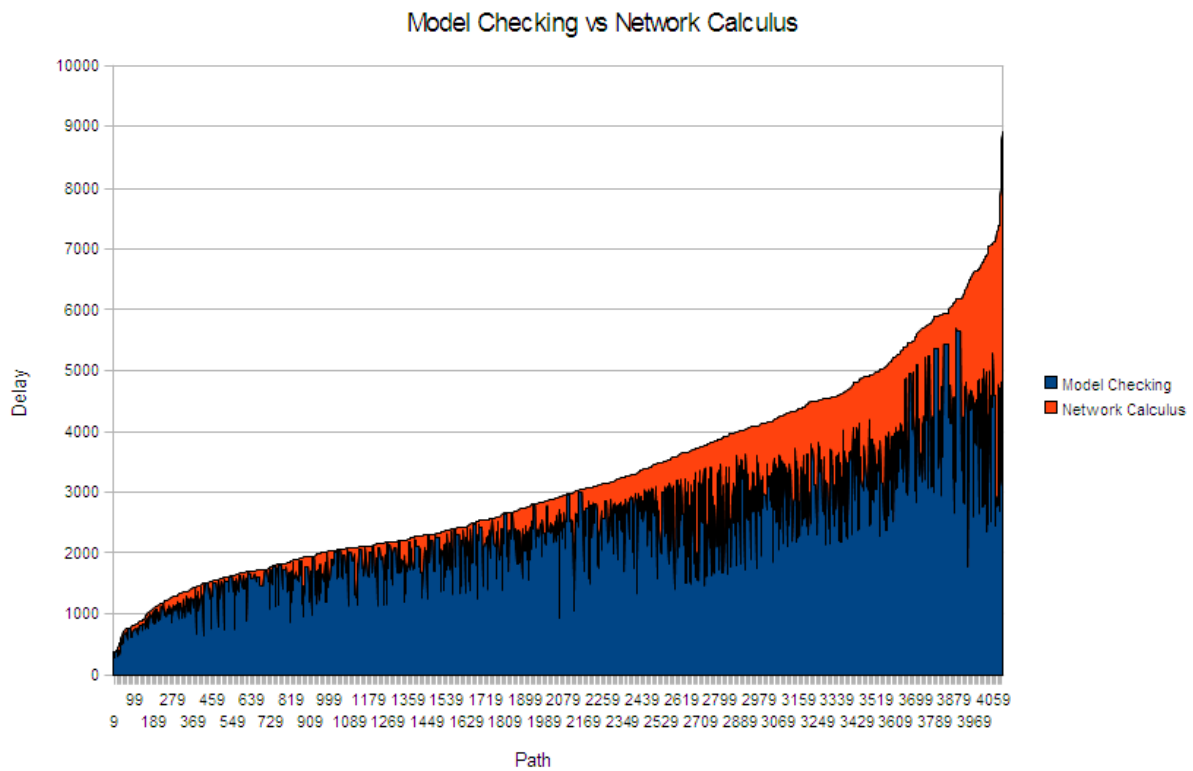


Figure 6.11 – Results of the case study compared to Network Calculus.

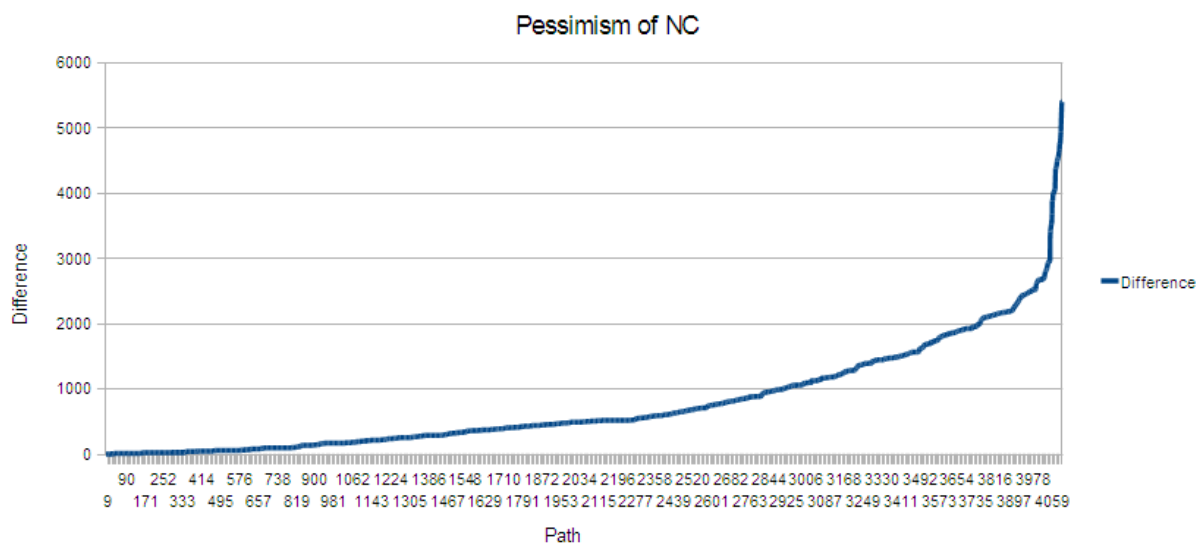


Figure 6.12 – Pessimism in Network Calculus.

Conclusions and Prospective

Contents

7.1	Conclusions	127
7.2	Prospective	129

7.1 Conclusions

An avionics system is an embedded system composed of communicating resources, sensors and actuators. Each communicate using shared communication networks such as the AFDX network. For certification reasons, the end to end communication delays need to be guaranteed. To ensure this property, several methods can be used:

- analytical methods, such as Network Calculus or Trajectory Approach, which compute an upper bound of the worst-case end-to-end transmission delays;
- model-checking and exhaustive simulation which evaluate exact worst-case end-to-end transmission delays.

Analytical methods can evaluate an industrial AFDX configuration but are pessimistic. In [Charara 2006a], Model-Checking has been used to compute the exact worst-case end-to-end delays of a small AFDX configuration. The method used is not well-adapted.

In this thesis, the goal is to compute the **exact** worst-case end-to-end delays on an industrial AFDX configuration. First, we propose to improve the Model-Checking approach by reducing the state-space in the verification process. The first contribution of the thesis is the Critical Instant Property.

Critical Instant Property. The worst-case waiting time of a packet in the output port of an AFDX switch occurs when the packet arrives at the output port at the same time of the others packets from others input links and when the packet is transmitted at last.

Considering this property, a timed automata modelling has been done. The method is able to analyze AFDX networks with up to 32 communicating flows, *i.e.* the virtual links in AFDX terminology, with support of periodic and sporadic data. Unfortunately, the method cannot handle an industrial AFDX network which is composed of more than 1000 VLs. The main reason is that the timed automata language is not well-adapted in managing large number of messages scheduled using FIFO queues and needs to be improved.

In the Model-Checking approach, we construct timed automata generating sequences of VLs which are candidate for worst-case scenario. The second contribution of this thesis is to **automatically** generate these sequences. The number of sequences is huge and we consider some properties of the AFDX to reduce this number. First, we define the Sequence Periodicity Property.

Sequence Periodicity Property. Since the sequences at the input port of the switches are periodic, it has been demonstrated that the sequences transmitted at the output port is periodic. The periodicity is equal to the hyper-period.

Considering this property, the sequence generation is considered only on two hyper-periods. So, the number of generated sequences is finite. The Critical Instant Property can be applied on these sequences, drastically reducing the search-space in the approach. A system with at least hundred VLs can be analyzed with these two properties.

To further reduce the state-space, 2 other AFDX properties has been defined:

Packet Ordering. If two packets with different amount of data arrive at the same instant in an output port of a switch, then the transmission of the packet with lesser data first, will induces an idle time between the packets. Otherwise, the transmission of the packet with the more amount of data first, induces no idle time.

Idle Time Placement in input links. As data rate of the input links is the same as data rate of the output links, the amount of data which is received by a switch directly impacts the amount of data which is stored into the output port. An idle time in the input links will reduce this amount which has to be stored and then will reduce the waiting time of the packets.

The goal in finding worst-case scenario is to order packets avoiding idle time in the input port of the switches. Finally, a reduced set of sequences which are candidates to the worst-case scenarios can be obtained. This set is composed of sequences for which the packet under study potentially faces the worst-case waiting time in the output ports of the switches by considering the ordering of the packets and by considering the critical instant property.

Computing the worst-case end-to-end transmission delays consists in finding the sequence with the highest transmission delay. In other words:

To compute the exact worst-case end-to-end transmission delays:

- we construct a reduced set of sequences which are candidates for the worst-case scenario;
- we choose the sequence which has the highest transmission delay among above set of scenarios.

The method has been applied to an industrial scale AFDX network composed of about 1000 VLS sharing more than 6000 paths. To make the computation, a tool has been developed. We are able to analyze all the VLS of this industrial AFDX network but not all the paths. We can analyze more than 60% paths. The obtained results have been compared with those obtained by analytical methods such as Network Calculus and Trajectory Approach. The real pessimism of these methods can also be evaluated thanks to exact worst case delays calculated by using our approach.

7.2 Prospective

Improvement of the approach. The AFDX communication model is complex. The methods used to analyze it need to define properties to break this complexity. In this thesis, 4 properties have been defined. But, when the method is applied to an industrial scale AFDX architecture, computation time is high because the number of sequences are still huge and hence

are difficult to analyze in short time. The solution proposed in this thesis is to stop the analysis after a given duration. So, for the VLs which do not finish in given time, the obtained results are optimistic.

The problem is due to the number of crossed flows in the system: the VLs which directly influence the VL under study and those which indirectly influence the VL under study. Both of them are considered in our approach. But, not all the indirect VLs have an effect on the worst-case end-to-end communication delay for a given path. The goal will be to find the characteristics of the VLs which do not influence the VL under study. By doing this, we will be able to reduce the number of VLs which need to be considered in our analysis and so, to reduce the number of generated sequences. This may eventually enable us to analyze all paths of an industrial scale AFDX network.

Generalization of the method to other Ethernet based networks. This work can also be extended to other networks and fields. For example, this work can be adapted for any switched Ethernet network with very minimum modifications, if any. Techniques presented in this work can be extended to space-wire communication protocol as well. This work can be applied to multi-processor distributed systems, where we can find inter node communication delays or even distributed scheduling.

Integration of the method in the global avionics system. Nowadays, the avionics systems are defined considering the Integrated Modular Avionics (IMA) architecture. They are composed of a set of applications which share a set of computing resources called modules communicating using a shared network, where network itself is connected to a set of sensors and actuators. The execution model is defined by the standard ARINC653 [ARINC 653 1997], while the communication standard is ARINC664 (part 7) [ARINC 664 2005], also known as AFDX. These architectures have been defined to allow some real-time requirements.

The goal of this thesis is to propose an approach to verify that the AFDX requirements are guaranteed. Some other works have been devoted to verify that the functional requirements are respected. As an example, in [Lauer 2010], timing analysis of functional chains on an IMA architecture has been done. The transmissions on the AFDX networks has been modeled by timed intervals obtained by using pessimistic upper bounds of Trajectory Approach. Such an analysis can be done by considering a global approach composed of functional models, such as the ones defined by [Lauer 2010], and the network model defined by our approach.

Another key area of research in IMA is the impact of spatial and temporal integration

choices on the communications performance [Nesrine 2013]. In the design of an IMA system, the problem is to allocate the partitions to the computation modules (spatial allocation) and to allocate the APEX communication channels to the various communications taking place between the tasks (temporal allocation). This integration has to guaranty that end to end communication delays are within allowed bounds. Using our models and methodology for the communication network, temporal allocation of IMA tasks can be integrated with spatial allocation algorithms and models. This can result in a tool that can be used to calculate the real end to end communication statistics for a modelled software and modelled physical architecture of an IMA system.

Model Checking Overview

Contents

A.1 Classification	133
A.2 List of Model Checkers, Modeling Languages and Specification Languages	135
A.2.1 List of Modeling Languages	135
A.2.2 List of Property Specification Languages	138
A.3 Relevance/Application to AFDX Network	138

The basic idea of model checking is to use algorithms, executed by computer tools, to verify the correctness of systems. The user inputs a description of a model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the verification up to the machine. If an error is recognized the tool provides a counter-example showing under which circumstances the error can be generated. The algorithms for model checking are typically based on an exhaustive state space search of the model of the system: for each state of the model it is checked whether it behaves "correctly", that is, whether the state satisfies the desired property. The tools used for this purpose (model checkers) can be classified in different groups depending on the way they work, or depending on the type of systems they can analyze.

A.1 Classification

Depending upon the type of system under consideration, the model checking tools can be classified as follows, this classification is generic in nature and there are model checkers which can be used for more than one category:

- *Real Time*: these model checkers are used for real-time systems where systems are constrained by operational deadlines from “event” to “system response”. In these model checkers time is an important variable and continuously evolves independent of other factors.
- *Hybrid*: these model checkers are used for hybrid systems where systems behavior exhibits both discrete and continuous change.
- *Probabilistic*: these model checkers are used for probabilistic systems where system exhibits inherent uncertainty which can be expressed in terms of probability such as randomized distributed algorithms, fault-tolerant processes and communication networks.
- *Simply Timed*: these model checkers are used for a special class of real-time systems where time evolves in discrete steps.
- *Un-timed (Plain)*: these model checkers are used for systems which don’t fall in any of the above category and are mostly used for verification of algorithms, protocols and logic.

Depending upon the way how model checkers work, there are basically two approaches in model checking that differ in the way the desired behavior, i.e. the requirement specification, is described:

- *Logic-based or heterogeneous approach*: in this approach the desired system behavior is captured by stating a set of properties in some appropriate logic, usually some temporal or modal logic. A system usually modeled as a finite-state automaton, where states represent the values of variables and control locations, and transitions indicate how a system can change from one state to another, is considered to be correct with respect to these requirements if it satisfies these properties for a given set of initial states.
- *Behavior-based or homogeneous approach*: in this approach both the desired and the possible behavior are given in the same notation (e.g. an automaton), and equivalence relations (or pre-orders) are used as a correctness criterion. The equivalence relations usually capture a notion like “behaves the same as”, whereas the pre-order relation represents a notion like “behaves at least as”. Since there are different perspectives and intuitions about what it means for two processes to “behave the same” (or “behave at least as”), various equivalence (and pre-order) notions have been defined. One of the most well-known notions of equivalence is bi-simulation. In a nutshell, two automata are bi-similar if one automaton can simulate every step of the other automaton, and vice versa. A frequently encountered notion of pre-order is (language) inclusion. An automaton A is included in automaton B , if all words accepted by A are accepted by B . A system is considered to be

correct if the desired and the possible behavior are equivalent (or ordered) with respect to the equivalence (or pre-order) attribute under investigation.

A.2 List of Model Checkers, Modeling Languages and Specification Languages

There are numerous model checkers for different types of systems. Some of them are still a work in progress and some are well established and proven softwares used in industry for certification purposes. Figure A.1 ¹ summarizes the currently available model checkers, it's not a comprehensive list of all model checkers but an effort to include important model checkers for comparison. It also enlists the languages used by these model checkers for modeling the systems and the languages used for specifying and verifying the properties.

A.2.1 List of Modeling Languages

- AltaRica: a language designed to model both functional and dysfunctional behaviours of critical systems.
- Cadence SMV: Cadence SMV Input Language; synchronous modeling language that has features supporting SMV's style of compositional refinement verification and abstract interpretation.
- CCS: Calculus of communicating systems; process calculus introduced by Robin Milner around 1980 and the title of a book describing the calculus.
- CCSP: A process calculus obtained from CCS by incorporating some operators of CSP. It is defined by Olderog [4] and by van Glabbeek/Vaandrager [5].
- CSP: Communicating sequential processes; formal language for describing patterns of interaction in concurrent systems. FDR2 is a refinement checking tool for CSP, comparing two models for compatibility.
- DVE input language: a system is described as Network of Extended Finite State Machines communicating via shared variables and unbuffered channels. Does not contain support for buffered channels and for checking the type of message to be received without performing the receive proper.

¹http://en.wikipedia.org/wiki/List_of_model_checking_tools

Comparison of some model checking tools

Name	Model Checking			Equivalence checking		GUI			Availability		
	Plain, Probabilistic, Stochastic, ...	Modelling language	Properties language	Supported equivalences	Counter example generation	GUI	Graphical Specification	Counter example visualization	Software license	Programming language used	Platform / OS
APMC (http://sylvain.berbiqui.org/apmc/download)	Approximate Probabilistic	Reactive modules	PCTL, PLTL		No	Yes	No	No	FUSC	C	Unix & related
ARC (http://altirica.labri.fr/wiki/tools:arc)	Plain	AltaRica	mu-calculus		No	No	No	No	FUSC	ANSI C	Unix & related
BANDERA (http://bandera.projects.cis.ksu.edu/)	Code analysis	Java	CTL, LTL		Yes	Yes	Yes	Yes	Free	Java	Windows and Unix related
BLAST (http://www.eecs.berkeley.edu/~tah/blast)	Code analysis	C	Monitor automata		Yes	No	No	No	Free	OCaml	Windows and Unix related
CADENCE SMV (http://www-cad.eecs.berkeley.edu/~kenmcmil/smv)	Plain	Cadence SMV, Verilog	CTL, LTL		Yes	Yes	No	No	FUSC	?	Windows and Unix related
CADP	Probabilistic	LOTOS	AFMC	SB, WB, BB, OE, STE, WTE, SE, tau*E	Yes	Yes	Yes	Yes	FUSC	?	MacOS, Linux, Solaris, Windows
CWB-NC (http://www.cs.sunysb.edu/~cwb/)	Plain and Timed	CCS, CSP, LOTOS, TCCS	AFMC, CTL, GCTL	SB, WB, me, ME	Yes	Yes	No	No	FUSC	SML of New Jersey	Windows and Unix related
DBRover (http://www.dbrover.com/)	Timed	Ada, C, C++, Java, VHDL, Verilog	LTL, MTL		No	Yes	Yes	Yes	Non-freeCommercial use only	?	Windows and Unix related
DiVinE Tool (http://anna.fi.muni.cz/divine)	Plain	DVE input language	LTL		No	Yes	No	No	Free	C/C++	Unix related
DREAM (http://dre.sourceforge.net/)	Real-time	C++, Timed automata	Monitor automata		Yes	No	No	No	Free	C++	Windows and Unix related
Edinburgh CWB (http://www.lfcs.ed.ac.uk/cwb)	Plain	CCS, TCCS, SCCS	Mu calculus	SB, WB, BB, me, ME, OE	Yes	No	No	No	FUSC	SML	Windows and Unix related
Expander2 (http://ls5-www.cs.uni-dortmund.de/~peter/Expander2/Expander2.html)	Hybrid		AFMC, CTL	SB, OE	No	Yes	No	No	Free	O'Haskell	Unix related
Fc2Tools (http://www-sop.inria.fr/meije/verification/)	Plain	FC2	?	SB, WB, BB	Yes	No	Yes	Yes	Free	?	Unix related
GEAR (http://jabc.cs.tu-dortmund.de/modelchecking/)	Plain	?	AFMC, CTL, mu-calculus		Yes	Yes	Yes	Yes	Free	Java	Windows and Unix related
LTSA (http://www.doc.ic.ac.uk/~jnm/book/)	Plain	FSP	LTL		Yes	Yes	No	Yes	Free	?	Windows and Unix related
MCMAS (http://www-lai.doc.ic.ac.uk/mcmas/)	Plain, Epistemic	ISPL	CTL, CTLK		Yes	Yes	No	Yes	Free	C++	Unix, Windows, Mac-OS
mCRL2	Real-time	mCRL2	mCRL2 mu-calculus	SB, BB, t*E, STE, WTE	Yes	Yes	No	Yes	Free	C++	MacOS, Linux, Solaris, Windows
MRMC	Real-time, Probabilistic	Plain MC	CSL, CSL, PCTL, PRCTL	SB	No	No	No	No	Free	C	Windows, Linux, MacOS
PRISM	Probabilistic	PEPA, PRISM language, Plain MC	PLTL, PCTL		No	Yes	No	No	Free	C++, Java	Windows, Linux, MacOS
Reactis Tester (http://www.reactive-systems.com/)	Hybrid	Simulink/Stateflow	?		No	Yes	Yes	No	Non-freeCommercial use only	?	Windows, Linux
RED (http://sourceforge.net/projects/redlib/)	Plain	TCTL	?		No	Yes	Yes	No	Non-freeCommercial use only	?	Linux
SPIN	Plain	Promela	LTL		No	Yes	No	No	FUSC	C, C++	Windows and Unix related
TAPAs	Plain	CCSP	CTL, mu calculus	SB, WB, BB, STE, WTE, me, ME, OE	Yes	Yes	Yes	Yes	Free	Java	Windows, MacOS and Unix related
UPPAAL	Real-time	Timed automata, C subset	TCTL subset		No	Yes	Yes	Yes	FUSC	C++, Java	Windows, Linux

Figure A.1 – Comparison of Model Checking tools.

A.2. List of Model Checkers, Modeling Languages and Specification Languages

- FC2: Machine-level ASCII representation for synchronized (hierarchical) networks of automata. Defined by the Esprit Basic Research Action CONCUR, 1992. Used as an input and exchange format by a number of verification tools, mainly in the area of process algebras.
- FSP: Finite State Processes.
- Java: Object-oriented programming language.
- LOTOS: Language Of Temporal Ordering Specification (ISO standard 8807); formal specification language based on temporal ordering used for protocol specification in ISO OSI standards.
- PEPA: Performance Evaluation Process Algebra; it is a stochastic process algebra designed for modelling computer and communication systems.
- Plain MC: these are simple text-file formats used in MRMC and PRISM.
- PRISM language: PRISM model description language.
- Promela: Process or Protocol Meta Language; it is a verification modeling language. The language allows for the dynamic creation of concurrent processes to model, for example, distributed systems.
- Reactive modules: Component-based modeling language for synchronous and asynchronous hardware and software systems.
- REDLIB: Timed CTL.
- Simulink/Stateflow: interactive design and simulation tool for event-driven systems.
- SCCS: Synchronous calculus of communicating systems.
- SMV: SMV input language.
- TCCS: Timed CCS.
- Verilog: an hardware description language (HDL) used to model electronic systems.
- VHDL: commonly used as a design-entry language for field-programmable gate arrays and application-specific integrated circuits in electronic design automation of digital circuits.
- Verus: A C like language used with Verus model checker

A.2.2 List of Property Specification Languages

- AFMC: Alternation Free Modal mu-Calculus.
- CSL: Continuous Stochastic Logic, characterizes bisimulation of continuous-time Markov processes.
- CSRL: Continuous Stochastic Reward Logic; a logic to specify measures over CTMCs extended with a reward structure (so-called Markov reward models).
- CTL: Computation Tree Logic; a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be an actual path that is realized.
- GCTL: Generalized Computation Tree Logic, it's both state based and action based.
- LTL: Linear temporal logic; a modal temporal logic with modalities referring to time.
- Monitor automata: Not sure about.
- mCRL2 mu-calculus: Kozen's propositional modal mu-calculus (excluding atomic propositions), extended with: - data-depended processes - quantification over data types - multi actions - time - regular formulas.
- mu-calculus: temporal logics with a least fix-point operator μ .
- PCTL: Probabilistic CTL; an extension of CTL which allows for probabilistic quantification of described properties.
- PLTL: Probabilistic Linear Temporal Logic.
- PRCTL: Probabilistic Reward Computation Tree Logic; it extends PCTL with reward-bounded properties.
- TCTL: Timed Computation Tree Logic is simply timed variant of CTL and is used in TSMV model checker.

A.3 Relevance/Application to AFDX Network

As described in Chapter 1, AFDX network falls under the category of a real-time system. Therefore, to analyze and model AFDX network we need to use a model checker that can handle real-time systems. As shown in Figure A.1, not all the model checkers support real-time

systems natively. This limits our choice to only fewer model checkers. There are other good model checkers under the category of “un-timed” or “Plain”, such as SMV, which we might like to use for AFDX network. These non real-time model checkers can also be used for real-time systems with a special modeling technique generally known as “Explicit-time Description Methods”. Explicit-Time Description Methods aim to verify real-time systems with general un-timed model checkers. Lamport [Lamport 2005] presented an explicit-time description method using a clock-ticking process (Tick) to simulate the passage of time together with a group of global variables for time requirements. Other techniques exist as well which don’t use global variables (*place reference of papers related to explicit time such as Verifying Real-Time Systems using Explicit-time Description Methods by Hao Wang and Wendy MacCaull*). The main idea of such techniques is to count clock ticks or time in the model. These techniques can only be used for simply timed systems that is where time is discrete. As the system model explicitly models the time, the obvious draw back is that state space increases rapidly and hence limits this method for smaller system models.

Software Architecture

Contents

B.1 Software Architecture	141
B.1.1 Parser	142
B.1.2 Network Pruning	143
B.1.3 Load Balancer	144
B.1.4 Compute Module	145
B.1.5 Control Logic	146

Based upon the algorithms and properties presented in this thesis, a software tool has been developed to compute exact worst case end to end communication delays of an AFDX network. The software is written in Java. Java was selected for its ability to be portable across multiple operating systems. The tool developed during this research work can be used as stand alone application on a single machine or as a distributed computation application running on multiple machines connected via network. There are two separate versions of the software which both have same algorithms for computations of end-to-end delays but differ in the way these calculations are managed: in one version computations runs on a single machine while in the other one computations are distributed across multiple machines. In this appendix, the software will be discussed in detail.

B.1 Software Architecture

The basic architecture of the software is shown in figure B.1. *Parser* is used to read the AFDX network configuration data and to initialize the internal constructs and variables. *Network Pruning* block builds the essential part of the network that is directly and indirectly linked with the VL under study and removes the *unconnected* part of the whole network. This block is also

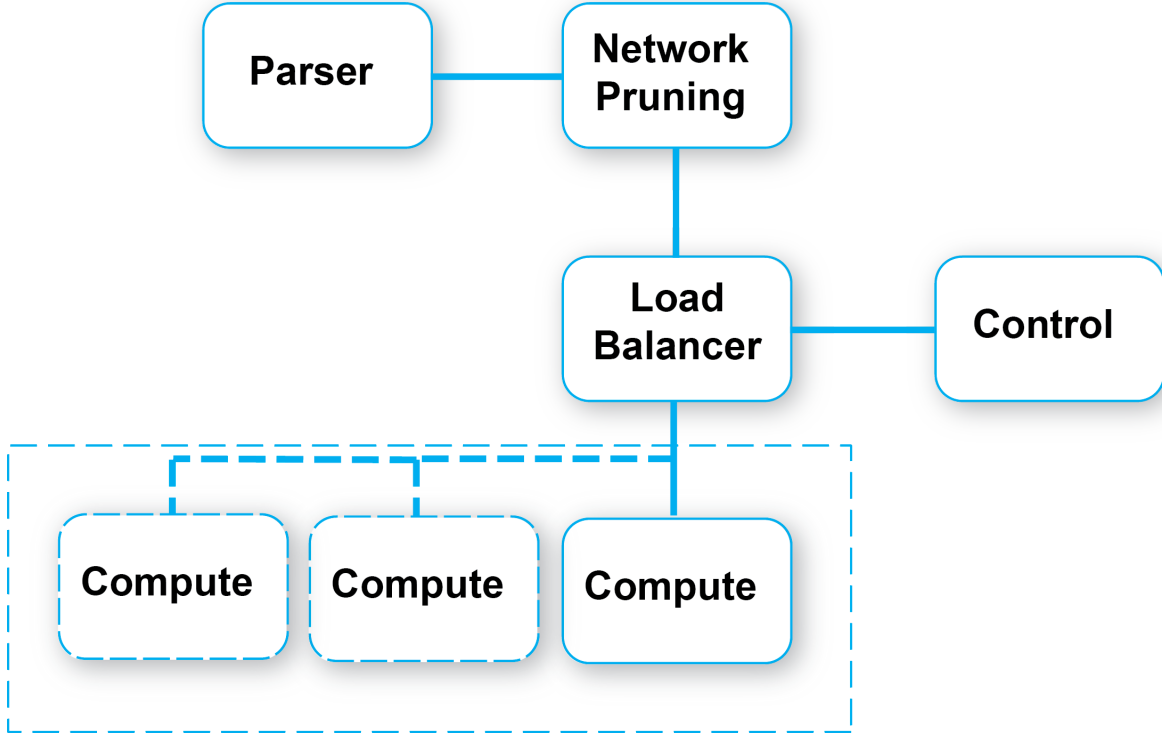


Figure B.1 – Software architecture.

responsible to generate scenarios which can be candidate for worst case end-to-end delays. In case of distribution computation, *Load Balancer* manages the computation distribution among available machine nodes. In case of single machine, it distributes computations among locally available *cores*. *Compute* module computes the delays and backlogs for a given sequence on a given output port. It also constructs set of resulting output sequences. Finally the *Control* module is responsible for coordination between all modules and for collection of results. The software uses *Flow Based Programming* [Morrison 2010] concepts to implement data flow approach where computations proceed as the data flow from one output port to another. For distributed computations, Java based library named *Java Parallel Processing Framework*(JPPF) is being used. The library can be found at <http://www.jppf.org/>

B.1.1 Parser

The input for this software is AFDX network configuration data. This data is in a specific format which is used by Airbus. In order to read this information encoded in specific format, a Parser is used in this software that takes a file as input and reads the configuration data in a

defined format from the input file. We are interested in following information from this file:

- Name of all the VLs
- Paths of each VL
- BAG value for each VL
- Maximum packet size for each VL

Currently, there is no offset information for strictly periodic VLs in this configuration file. The offset assignment for strictly periodic VLs can be added to configuration file very easily. Presently, we assume no offsets for strictly periodic VLs but if required it can be assigned easily with a variable already defined in the data structure code of the parser. Parser will parse all above info from the configuration file and store it in the data structures for the use by rest of the software. The output of the parser consists of list of all paths of the AFDX network with associated data *i.e.* VL name, BAG value, maximum packet size etc. for each path.

B.1.2 Network Pruning

After parsing the AFDX network configuration file, we have list of all the paths in the network. At this stage, we need the reference path *i.e.* the path which is under study for end to end communication delays. This path can be defined in many ways: either directly taken from user input, or defined in code, or in an input file. Once we have the reference path, we need to find the portion of network which is relevant to path under study *i.e.* the part of network which is connected either directly or indirectly with the path under study. This is necessary in order to reduce the complexity involved in computations. We refer this step as *Network Pruning*. After network pruning, the remaining part of the network contains only the VL paths which we consider in our algorithms.

From the list of remaining paths, after network pruning, we generate list of possible scenarios which can be candidate for worst case end to end delays. This list is generated on the fly, *i.e.* we do not construct or store the whole list in memory but we do it one case at a time. This is achieved thanks to lists of packets in each end system which is arranged according to properties we have developed in Chapter 5. This is explained in section 5.5.4.

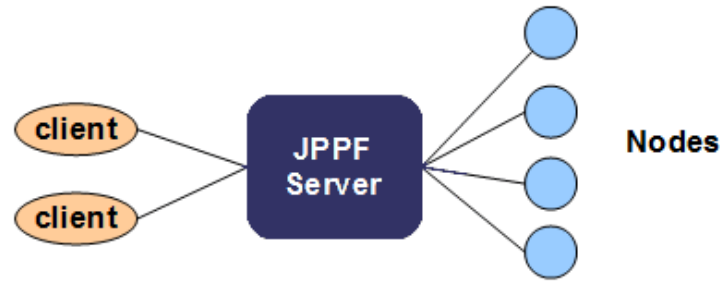


Figure B.2 – JPPF Grid architecture.

B.1.3 Load Balancer

As mentioned earlier, this software has two versions: one which supports distributed computations and the one which runs on single machine only. The purpose of load balancer is to distributed overall computation load to available resources. For distributed version, free java library *Java Parallel Processing Framework*(JPPF) is being used to spread the computations over available nodes connected through a network. JPPF makes it easy to parallelize computationally intensive tasks and execute them on a grid. JPPF is based on client-server architecture.

A JPPF grid is made of three main components that communicate together:

- Clients are entry points to the grid and enable developers to submit work via the client APIs
- Servers are the components that receive work from the clients, dispatch it to the nodes, receive the results from the nodes, and send these results back to the clients
- Nodes perform the actual work execution

The figure B.2 below shows how all the components are organized together. From this picture, we can see that the server plays a central role, and its interactions with the nodes define a master / slave architecture, where the server (i.e. master) distributes the work to the nodes (i.e. slaves). Though here we have shown only one server, but JPPF allows more than one servers on a grid interconnected by peer-to-peer network.

The smallest unit of work that can be handled in a JPPF grid is known as *Task*. A logical grouping of all the tasks submitted together is known as *Job*. So in order to use JPPF grid, in our software, we must define task. For this purpose, we consider that computation of delay on a single output port is the smallest unit of work that can be distributed and hence it is considered

as a task. The detail of what this task does is explained in section 5.2.1. The sequence of computations is given as:

- Take the total number of cases which are candidate for worst case scenario
- For each case, on data flow basis, submit the task of calculation of delay on a single port to the JPPF server. Once the results are obtained, submit the task of computation in the consequent port in the path of considered case till all the output ports have been computed.
- submit the results back to the client which gathers results.
- Once all cases have been computed, store/print the worst case and associated end to end communication delay.

Load balancing is handled by JPPF server. More details of how JPPF server handles and manages the load on a grid, please consult the online manual at <http://www.jppf.org/>

For stand alone version, we use number of threads equal to number of cores available on the machine, and compute one case in one thread till all the cases have been analyzed. gathering of results is similar to distributed version: we gather all the results and then store/print the worst case and associated end to end communication delay. In both versions, we can store the results of all cases but in order to save memory usage, we only store the result if its worse than existing result.

B.1.4 Compute Module

This is the module where each *task* is executed. In this module we compute the delay on a given output port and obtain the resulting sequences. Algorithm of section 5.2.1 is implemented in this module. In distributed version of the software, this piece of code is executed on the grid. Input to this module is sequences at the given port and output from this module is the resulting delay at this port for the input sequences along with output sequence obtained after merging the input sequences, as shown in figure 5.3. We implement the merging of sequence by using *Array*. Each row of the array represents a packet in the sequence. Columns of the array represent information about the packet. Packets of first sequence are stored in array as they appear in the sequence. For remaining sequences, we check if the packet to be added in the array overlaps with existing packet or not. In case it does not overlap with existing packet, then

it is stored in Array at the current position. If packet overlaps with the existing packet in the array, we move the packets depending upon their time of arrival.

B.1.5 Control Logic

Control module handles the overall coordination among modules and user interactions. In distributed version, this module acts as client. This module gathers the results and saves final result on disk as well. Control module acts as a glue logic among other core modules. For example, Control module takes the input AFDX configuration file and passes over to Parser. Then takes output from Parser and given it to Network Pruning module etc. In this module we also define the path under study. For analyzing the complete network of the case study in 6, we use a loop. In this loop: we define first path of the network as a reference path, then compute the worst case end to end communication delay for this reference path, change the reference path to the next path in the total paths of the network and repeat the loop till we have analyzed all the paths of the network.

Bibliography

- [Adnan 2010a] Muhammad Adnan. State of the art in model checking: with application to afdx network. Master thesis, ENSEEIHT Toulouse, 2010. (Cited on pages 5, 41 and 50.)
- [Adnan 2010b] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. *Model for worst-case delay analysis of an AFDX network using timed automata*. In Proc. of the 15th ETFA (WiP session), Bilbao, Septembre 2010. (Cited on pages 3, 5, 50 and 111.)
- [Adnan 2010c] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. *Worst-case end-to-end delay analysis of switched Ethernet using timed automata*. In Junior Researcher Workshop on Real-Time Computing , Toulouse, pages 23–26, <http://www.irit.fr/>, November 2010. IRIT. (Cited on page 5.)
- [Adnan 2011a] Muhammad Adnan, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. *An improved timed automata model for computing exact worst-case delays of AFDX periodic flows*. In Proc. of the 16th ETFA (WiP session), Toulouse, Septembre 2011. (Cited on pages 5, 76, 77 and 79.)
- [Adnan 2011b] Muhammad Adnan, Jean-Luc Scharbarg and Christian Fraboul. *Minimizing the search space for computing exact worst-case delays of AFDX periodic flows*. In Proc. of the 6th SIES, Vasteras, June 2011. (Cited on pages 5 and 58.)
- [Adnan 2012] M. Adnan, J.-L. Scharbarg, J. Ermont and C. Fraboul. *An improved timed automata approach for computing exact worst-case delays of AFDX sporadic flows*. In Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on, pages 1–8, 2012. (Cited on pages 5 and 79.)
- [Alur 1994] Rajeev Alur and David L. Dill. *Theory of Timed Automata*. Theoretical Computer Science, vol. 126, no. 2, pages 183–235, 1994. (Cited on page 3.)
- [Apt 1986] K. R. Apt and D. Kozen. *Limits for the automatic verification of finite-state concurrent systems*. In Information Processing Letters, volume 22(6), pages 307–309, 1986. (Cited on page 15.)
- [ARINC 653 1997] Aeronautical Radio Inc ARINC 653. *ARINC Specification 653*. Avionics Application Software Standard Interface, 1997. (Cited on pages 15 and 130.)

- [ARINC 664 2005] Aeronautical Radio Inc ARINC 664. *AIRCRAFT DATA NETWORK PART 7 AVIONICS FULL DUPLEX SWITCHED ETHERNET (AFDX) NETWORK*. 2005. (Cited on pages 3, 18 and 130.)
- [Baier 2008] Christel Baier and Joost-Pieter Katoen. Principles of model checking (representation and mind series). The MIT Press, 2008. (Cited on pages 1, 2 and 12.)
- [Bauer 2009] Henri Bauer, Jean-Luc Scharbarg and Christian Fraboul. *Applying and optimizing Trajectory approach for performance evaluation of AFDX avionics network*. In Proc. of the 14th International Conference on Emerging Technologies and Factory Automation, pages 1–8, Mallorca, september 2009. IEEE. (Cited on pages 3 and 39.)
- [Bauer 2010] Henri Bauer, Jean-Luc Scharbarg and Christian Fraboul. *Improving the worst-case delay analysis of an AFDX network using an optimized trajectory approach*. IEEE transactions on industrial informatics, vol. 6, no. 4, pages 521–533, Novembre 2010. (Cited on pages 3, 39, 59 and 87.)
- [Bérard 2001] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci and Ph. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, 2001. (Cited on pages 15 and 79.)
- [Boehm 2001] Barry Boehm and Victor R. Basili. *Software Defect Reduction Top 10 List*. Computer, vol. 34, no. 1, pages 135–137, January 2001. (Cited on page 10.)
- [Boyer 2008] M. Boyer and C. Fraboul. *Tightening end to end delay upper bound for AFDX network calculus with rate latency FIFO servers using network calculus*. In Factory Communication Systems, 2008. WFCS 2008. IEEE International Workshop on, pages 11–20, 2008. (Cited on page 39.)
- [Charara 2006a] Hussein Charara, Jean-Luc Scharbarg, Jérôme Ermont and Christian Fraboul. *Methods for Bounding end-to-end Delays on an AFDX Network*. In Proceedings of the 18th ECRTS, pages 193–202, Dresden, Germany, July 2006. (Cited on pages 3, 62, 81, 87, 97, 111 and 127.)
- [Charara 2006b] Hussein Charara, Jean-Luc Scharbarg, Jerome Ermont and Christian Fraboul. *Methods for bounding end-to-end delays on an AFDX network*. ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems, pages 193–202, 2006. (Cited on pages 5, 50, 55 and 58.)
- [Clarke 1981] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. In In Logic of Programs, volume 131 of Lecture Notes in Computer Science, pages 52–71. Springer-Verlag, 1981. (Cited on page 15.)

- [Clarke 1996a] E. M. Clarke and R. Kurshan. *Computer-aided verification*. In IEEE Spectrum, volume 33(6), pages 61–67, 1996. (Cited on page 15.)
- [Clarke 1996b] E. M. Clarke and J. Wing. *Formal methods: state of the art and future directions*. In ACM Computing Surveys, volume 28(4), pages 626–643, 1996. (Cited on page 15.)
- [Clarke 1999] E. M. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, 1999. (Cited on page 15.)
- [Clarke 2000] E. M. Clarke and H. Schlingloff. *Model checking*. In In A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning (Volume II), pages 1635–1790. Elsevier Publishers B.V., 2000. (Cited on page 15.)
- [Cruz 1991a] R.L. Cruz. *A Calculus for network delay, Part I*. IEEE Transactions on Information Theory, vol. 37, no. 1, pages 114–131, January 1991. (Cited on page 3.)
- [Cruz 1991b] R.L. Cruz. *A Calculus for network delay, Part II*. IEEE Transactions on Information Theory, vol. 37, no. 1, pages 132–141, January 1991. (Cited on page 3.)
- [Discussion Group 2010] UPPAAL Discussion Group. <http://tech.groups.yahoo.com/group/uppaal/message/1609>. 2010. (Cited on pages 72 and 77.)
- [Fidge 2006] Colin Fidge and Yu-Chu Tian. *Functional Analysis of a Real-Time Protocol for Networked Control Systems*. In Susanne Graf and Wenhui Zhang, editors, Automated Technology for Verification and Analysis, volume 4218 of *Lecture Notes in Computer Science*, pages 446–460. Springer Berlin Heidelberg, 2006. (Cited on page 15.)
- [Fraboul 2002a] C. Fraboul and F. Frances. *Applicability of Network Calculus to the AFDX*. Rapport technique PBAR-JD-728.0821/2002, 2002. (Cited on pages 3 and 87.)
- [Fraboul 2002b] C. Fraboul and F. Frances. *Applicability of Network Calculus to the AFDX*. Technical Report PBAR-JD-728.0821/2002, 2002. (Cited on pages 26 and 31.)
- [Frances 2006] F. Frances, C. Fraboul and J. Grien. *Using network calculus to optimize the AFDX network*. In Proceedings of ERTS, Toulouse, France, January 2006. (Cited on pages 26 and 31.)
- [Hajek 1978] J. Hajek. *Automatically verified data transfer protocols*. In In 4th International Conference on Computer Communication (ICCC), pages 749–756. IEEE Computer Society Press, 1978. (Cited on page 15.)
- [Holzmann 1990] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1990. (Cited on page 15.)

- [Holzmann. 1994] G.J. Holzmann. *The theory and practice of a formal method: NewCoRe*. 13th IFIP World Computer Congress, North Holland, pages 35–44, 1994. (Cited on page 14.)
- [Huth 1999] M. Huth and M. D. Ryan. *Logic in Computer Science - Modelling and Reasoning about Systems*. Cambridge University Press, 1999. (Cited on page 15.)
- [Jean-Yves Le Boudec 2001] Patrick Thiran Jean-Yves Le Boudec. *Network calculus: A theory of deterministic queuing systems for the internet*. Springer-Verlag, Berlin, DE, 2001. (Cited on page 26.)
- [Kurshan 1994] R. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994. (Cited on page 15.)
- [Lamport 2005] Leslie Lamport. *Real-Time Model Checking Is Really Simple*. Lecture Notes in Computer Science, pages 162–175, 2005. (Cited on page 139.)
- [Lauer 2010] Michaë Lauer, Jérôme Ermont, Claire Pagetti and Frédéric Boniol. *Analyzing end-to-end functional delays on an IMA platform*. In Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part I, ISoLA'10, pages 243–257, Berlin, Heidelberg, 2010. Springer-Verlag. (Cited on pages 15 and 130.)
- [Le Boudec 2001] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001. ISBN: 3-540-42184-X. (Cited on page 3.)
- [Li 2010] Xiaoting Li, Jean-Luc Scharbarg and Christian Fraboul. *Improving end-to-end delay upper bounds on an AFDX network by integrating offsets in worst-case analysis*. In Proc. of the 14th ETFA, Bilbao, septembre 2010. (Cited on pages 3, 39 and 44.)
- [Martin 2004] Steven Martin. *Maîtrise de la dimension temporelle de la qualité de service dans les réseaux*. PhD thesis, Université Paris XII, July 2004. (Cited on page 32.)
- [Martin 2006a] S. Martin and P Minet. *Schedulability analysis of flows scheduled with FIFO: application to the expedited forwarding class*. In 20th International parallel and distributed processing symposium, Rhodes Island, Greece, April 2006. (Cited on pages 3, 32, 38 and 87.)
- [Martin 2006b] Steven Martin and Pascale Minet. *Worst case end-to-end response times of flows scheduled with FP/FIFO*. In 5th IEEE International Conference on Networking, Mauritius, April 2006. (Cited on page 32.)
- [McMillan 1993] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. (Cited on page 15.)

- [Merz 2001] S. Merz. *Model checking: a tutorial*. In In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modelling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001. (Cited on page 15.)
- [Migge 1999] Jorn Migge. *L’ordonnancement sous contraintes temps-réel : un modèle á base de trajectoires*. PhD thesis, INRIA, Sophia Antipolis France, November 1999. (Cited on page 32.)
- [Morrison 2010] J. Paul Morrison. *Flow-based programming, 2nd edition: A new approach to application development*. CreateSpace Independent Publishing Platform, 2010. (Cited on pages 120 and 142.)
- [Nesrine 2013] Badache Nesrine, Jaffres-Runser Katia, Jean-Luc Scharbarg and Christian Fraboul. *End to End delay analysis in an Integrated Modular Avionics architecture*. In *Proc. of the 18th ETFA (WiP session)*, Cagliari, Septembre 2013. (Cited on page 131.)
- [NuSMV] NuSMV. <http://nusmv.fbk.eu/>. (Cited on pages 41 and 43.)
- [Queille 1982] J.-P. Queille and J. Sifakis. *Specification and verification of concurrent systems*. In CESAR. In *5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982. (Cited on page 15.)
- [Ruel 2008] Silvain Ruel and FAURE Jean-Marc. *Building effective formal methods to prove time properties of networked automation systems*. In in *Proc. of 9th International Workshop on Discrete Event Systems*, Goteborg, Sweden, May 2008. (Cited on page 15.)
- [Ruys 2003] T. C. Ruys and E. Brinksma. *Managing the verification trajectory*. In *International Journal on Software Tools for Technology Transfer*, volume 4(2), pages 246–259, 2003. (Cited on page 15.)
- [Scharbarg 2009] Jean-Luc Scharbarg, Frédéric Ridouard and Christian Fraboul. *A probabilistic analysis of end-to-end delays on an AFDX network*. *IEEE transactions on industrial informatics*, vol. 5, no. 1, February 2009. (Cited on pages 3 and 58.)
- [Schneider 2004] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer-Verlag, 2004. (Cited on page 15.)
- [Steve 2007] LIMAL Steve and LESAGE Jean-Jacques. *Formal verification of redundant media extension of Ethernet PowerLink*. In *Proceedings of the ETFA*, Patras, Grece: IEEE, September 2007. (Cited on page 15.)
- [Straunstrup 2000] J. Straunstrup, H.R. Andersen, H. Hulgaard, J. Lind-Nielsen, G. Behrmann, K. Kristoffersen, A. Skou, HH. Leerberg and N.B. Theilgaard. *Practical verification of embedded software*. *Computer*, vol. 33, no. 5, pages 68–75, 2000. (Cited on page 13.)

- [Tindell 1994] Ken Tindell and John Clark. *Holistic schedulability analysis for distributed hard real-time systems*. Microprocess. Microprogram., vol. 40, no. 2-3, pages 117–134, 1994. (Cited on page 32.)
- [UPPAAL] UPPAAL. <http://www.uppaal.com>. (Cited on pages 41 and 49.)
- [West 1978] C. H. West. *An automated technique for communications protocol validation*. In IEEE Transactions on Communications, volume 26(8), pages 1271–1275, 1978. (Cited on page 15.)
- [West 1989] C. H. West. *Protocol validation in complex systems*. In In Symposium on Communications Architectures and Protocols, pages 303–312. ACM Press, 1989. (Cited on page 15.)
- [Whittaker 2000] J.A. Whittaker. *What is software testing? And why is it so hard?* Software, IEEE, vol. 17, no. 1, pages 70–79, 2000. (Cited on page 10.)
- [Wolper 1995] P. Wolper. *An introduction to model checking*. In Position statement for panel discussion at the Software Quality workshop, 1995. (Cited on page 15.)
- [Zhang 1995] Hui Zhang. *Service Disciplines for guaranteed performance service in packet-switching networks*. Proceedings of the IEEE, vol. 83, no. 10, pages 1374–1396, October 1995. (Cited on page 81.)

Index

ADN, [17](#)
AFDX, [2](#), [17](#)
ARINC 429, [17](#)
Arrival Curve, [25](#)

BAG, [20](#)
Busy period, [31](#)

Complexity of Java based tool, [93](#)
Complexity of timed automata, [73](#)
Concatenation of curves, [28](#)
CSMA/CD, [18](#)
Cumulative Functions, [25](#)

Emulation, [10](#)
Exhaustive Simulation, [9](#)

ICT, [1](#)
IMA, [14](#)
Input and Output Functions, [25](#)

Model, [11](#)
Model checker, [11](#)

Network Calculus, [24](#)
Network Calculus Bounds, [27](#)

Peer review, [9](#)

Service Curve, [25](#)
Simulation, [10](#)
State-Space, [12](#), [37](#)
Structural analysis, [10](#)

Trajectory approach, [30](#)

Virtual Link, [20](#)

List of Abbreviations

ADN Aircraft Data Network

AFDX Avionics Full-Duplex Switched Ethernet

ARINC Aeronautical Radio, Incorporated

ATM Asynchronous Transfer Mode

BAG Bandwidth Allocation Gap

BDD Binary Decision Diagram

CSMA/CD Carrier Sense Multiple Access with Collision Detection

CTL Computation Tree Logic

ES End System

ESA European Space Agency

FAA Federal Aviation Authority

FIFO First In First Out

FSM Finite State Machine

ICT Information and Communication Technology

IMA Integrated Modular Avionics

LTL Linear Temporal Logic

NASA National Aeronautics and Space Administration

NC Network Calculus

QoS Quality of Service

SW Switch

TA Timed Automata

VL Virtual Link

Exact Worst-Case Communication Delay Analysis of AFDX Network

Abstract: The main objective of this thesis is to provide methodologies for finding exact worst case end to end communication delays of AFDX network. Presently, only pessimistic upper bounds of these delays can be calculated by using Network Calculus and Trajectory approach.

To achieve this goal, different existing tools and approaches have been analyzed in the context of this thesis. Based on this analysis, it is deemed necessary to develop new approaches and algorithms.

First, Model checking with existing well established real time model checking tools are explored, using timed automata.

Then, exhaustive simulation technique is used with newly developed algorithms and their software implementation in order to find exact worst case communication delays of AFDX network.

All this research work has been applied on real life implementation of AFDX network, allowing us to validate our research work on industrial scale configuration of AFDX network such as used on Airbus A380 aircraft.

Keywords: AFDX Network, Model Checking, Worst Case Communication Delay, Exhaustive Simulation

Analyse du délai de transmission pire cas exact du réseau AFDX

Résumé : L'objectif principal de cette thèse est de proposer les méthodes permettant d'obtenir le délai de transmission de bout en bout pire cas exact d'un réseau AFDX. Actuellement, seules des bornes supérieures pessimistes peuvent être calculées en utilisant les approches de type Calcul Réseau ou par Trajectoires.

Pour cet objectif, différentes approches et outils existent et ont été analysées dans le contexte de cette thèse. Cette analyse a mis en évidence le besoin de nouvelles approches.

Dans un premier temps, la vérification de modèle a été explorée. Les automates temporisés et les outils de vérification ayant fait leur preuve dans le domaine temps réel ont été utilisés.

Ensuite, une technique de simulation exhaustive a été utilisée pour obtenir les délais de communication pire cas exacts.

Pour ce faire, des méthodes de réduction de séquences ont été définies et un outil a été développé.

Ces méthodes ont été appliquées à une configuration réelle du réseau AFDX, nous permettant ainsi de valider notre travail sur une configuration de taille industrielle du réseau AFDX telle que celle embarquée à bord des avions Airbus A380.

Mots-clés : Réseau AFDX, Vérification de modèles, Délai de communication pire cas, Simulation exhaustive
